SIEMENS

# ASSEMBLER (BS2000)

## Reference Manual

**Edition May 1987**
**(ASSEMBLER V29.1C)**

Order No. U62-J-Z55-6-7600
Printed in the Federal Republic of Germany
3030 AG 8870.8 (3790)

Order No.:
U62-J-Z55-6-7600

A ring binder for this manual is available at a cost of DM 4.20.
Order number: U1225-J-Z18-1

78689

# ASSEMBLER (BS2000)
# Reference Manual

Edition May 1987
(ASSEMBLER V29.1C)

Order No. U62-J-Z55-6-7600
Printed in the Federal Republic of Germany
3030 AG 8870.8 (3790)



Siemens Aktiengesellschaft

# PREFACE

This manual is intended for the assembly language programmer. It covers the

- Assembler environment,
- Assembly language,
- Macro language,
- Handling of the assembler,
- Messages, error messages and warnings of the assembler.

A description of the Post-Assembly Diagnostic Routine (ADIAG) is also included.

The present manual is designed as a reference book. The reader is supposed to be familiar with the assembly language.

The differences between this and the obsolete version are detailed in the List of Amendments.

Literature references are specified in the text in abbreviated form. The complete title of every publication referred to is contained in the list of "References". The corresponding System Reference Guide is listed in addition.

To help us continue improving our publications, please send us your comments, requests and suggestions using the reply forms provided at the back.

**Editorial Department D ST QM 2**

# LIST OF AMENDMENTS 1

**The following amendments have been made since publication of the Revision September 1985 (Assembler V29.1B), and incorporated in the Edition May 1987 (Assembler V29.1C)**

The most important amendments are:

- PLAM source member name > 8 characters
- Monitor job variables support
- EDT fullscreen mode (ADIAG)
- Revised description of dummy registers

The following table lists the amendments in detail:

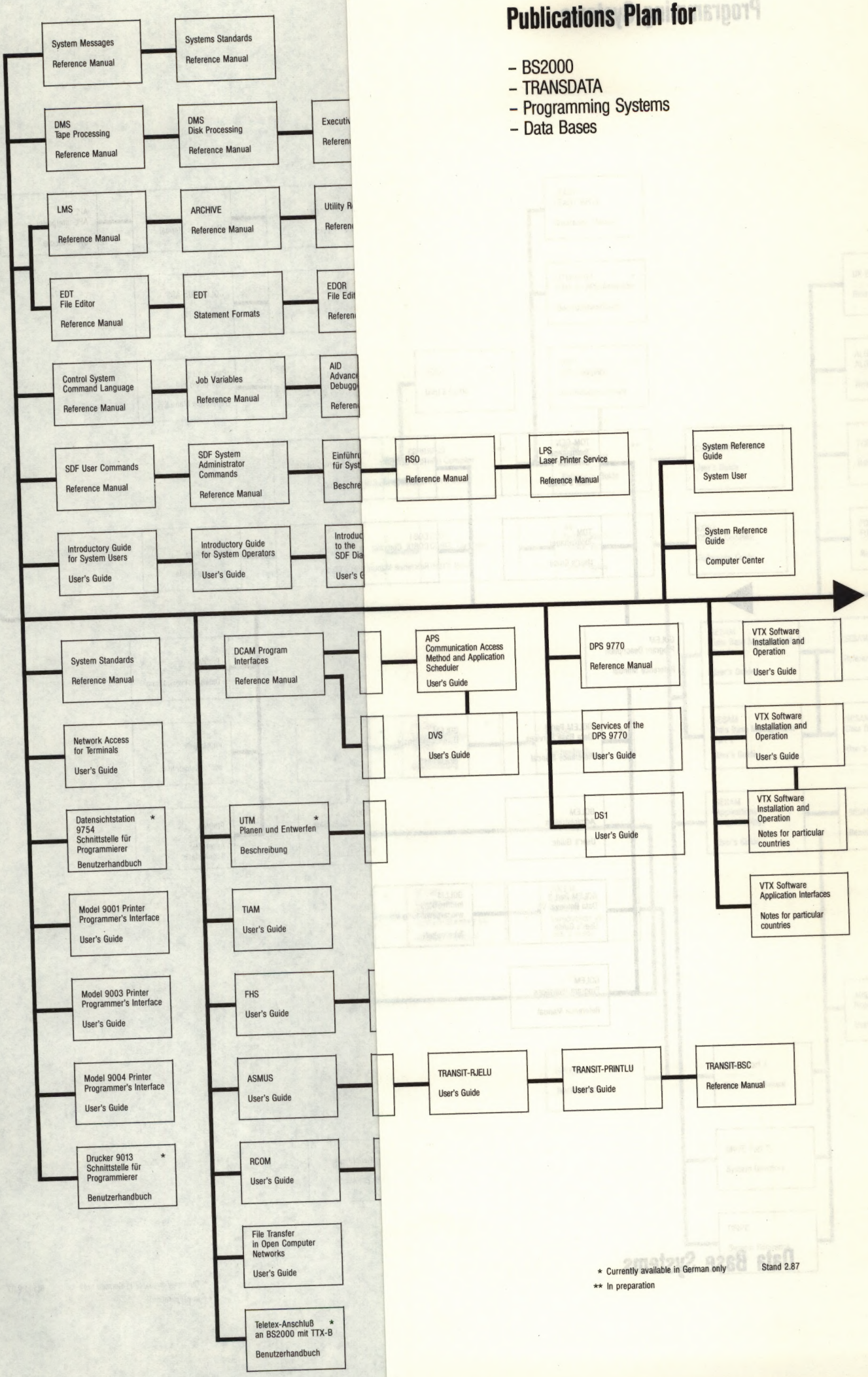
| Page | Topic | new | modi-fied | omit-ted |
|---|---|---|---|---|
| 2-2 | IOPACKAGE, IOSTART, ILA to be loaded unshared | | | X |
| 2-6 | Last paragraph | X | | |
| 3-1 | Reference to 4.7.4 | X | | |
| 3-4 | Reference to OPSYN | X | | |
| 3-17 | Reference to further examples | X | | |
| 3-18 | Reference to LTORG | X | | |
| 3-30/32 | Reference to 3.17 | X | | |
| 3-40 | Text added | X | | |
| 3-43 | Note on V-type address constant | | X | |
| 3-60 | Absolute value | | X | |
| 3-80 | Text changed in the last but one paragraph preceding the first example | | X | |
| 3-82 ff | New text for dummy registers | X | | |
| 3-90 | Text deleted: column 80 | | | X |
| 3-98 | Last 4 notes: new text added | X | | |

**List of amendments**

| Page | Topic | new | modi-fied | omit-ted |
|---|---|---|---|---|
| 4-1 | Change: There are four types of macro instructions | | X | |
| 4-2 | Notes on macro library | X | | |
| 4-5 | Text changed: Four formats | | X | |
| 4-10 | 4th paragraph: redefined | | X | |
| 4-14 | Error code ... 3 digits<br>MNOTE* | <br>X | X | |
| 4-16 | Four types of macro instructions | | X | |
| 4-23 | Reference to 4.7.4 | X | | |
| 4-28 | New text on system variable symbols | X | | |
| 4-36 | Reference to A.9, item 10 (Default Values) | X | | |
| 4-37 | Text amended in first paragraph | | X | |
| 4-68 | LCLA inserted | X | | |
| 5-3 | Note: Multiple start using SAVLST option | X | | |
| 5-6 | EDT full-screen mode | X | | |
| 6-2 | ATXREF + INSTR = SET1 | X | | |
| 6-5 | SAVLST, multiple start | X | | |
| 6-7 | "+" {Libr(name)} | | | X |
| 6-7/8 | SOURCE option remarks | X | | |
| 6-8 | Reference to LMS conventions | X | | |
| 6-10 | Note: Task switch as of 29.0 | | X | |
| 6-12 | /PARAM CARD=YES | | X | |
| 6-13 | Example 1 new, examples 2 + 3 amended | X | | |
| 6-15 | Text SOURCE PROGRAM | | X | |
| 6-17 | Error messages:<br>LOCKED, NEXT ATTEMPT AFTER 6 SECONDS<br>OPEN ERROR OD99 | <br>X<br>X | | |
| 6-20 | Text after SOURCE LIBRARY (7.) | X | | |
| 6-22 | Note on error flags | | X | |
| 6-22 ff | Monitor job variable support | X | | |
| A11-1ff | Appendix A.11: Dummy register examples | X | | |

# Publications Plan for

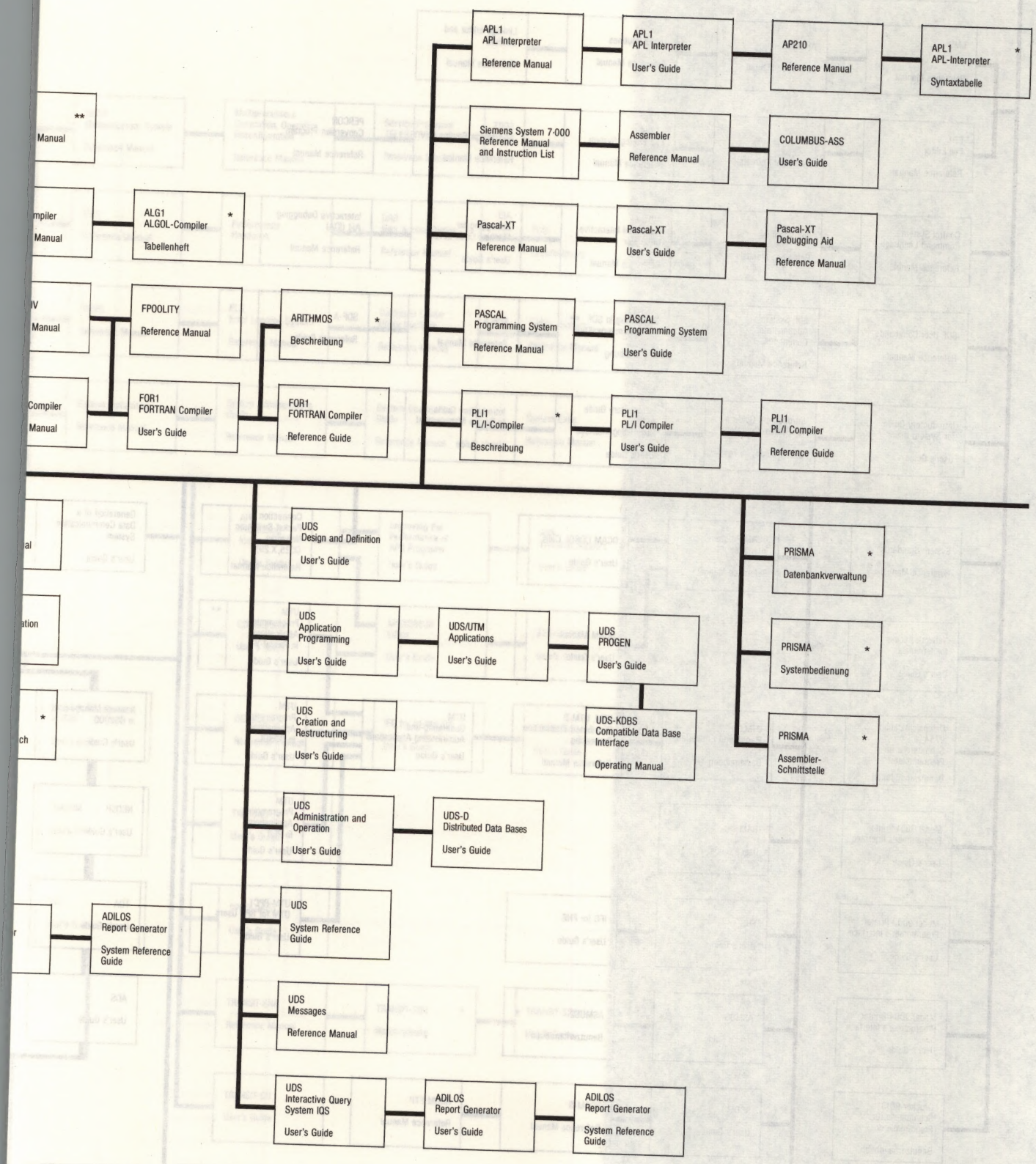
- BS2000
- TRANSDATA
- Programming Systems
- Data Bases




* Currently available in German only

** In preparation

Stand 2.87




* Currently available in German only

** In preparation

Stand 2.87

# CONTENTS

# 1 INTRODUCTION

The assembly language is a machine-oriented, symbolic programming language which expedites the writing of programs for execution in the operating system BS2000.

Machine instruction statements are a one-for-one symbolic representation of actual machine instructions. The assembler produces an equivalent machine instruction in the object program for each machine instruction statement in the source program.

Assembly instruction statements provide auxiliary functions that assist the programmer in:

- checking and documenting his programs,
- controlling the assigment of storage addresses,
- program sectioning and linking,
- defining data and storage fields,
- controlling the assembly system itself.

Assembly instruction statements specify the auxiliary functions to be performed during the assembly and, with a few exceptions, do not result in the generation of any machine language code during the assembly.

Macro instruction statements enable the assembler to retrieve, during the assembly, routines which have already been coded (written in assembly language), to modify these routines according to information supplied in the macro instruction, and insert the generated source statements in place of the macro statement.
The resulting source program is translated into machine language.

# 2 ASSEMBLER ENVIRONMENT



## 2.1 CONFIGURATION

The minimum configuration for running the assembler is the same as the minimum specified for the Executive.

In general, the assembler is file-oriented. Utility files are used as the assembly work devices.

## 2.2 BATCHED ASSEMBLIES

Several assemblies may be processed once the assembler has been loaded. Multiple assemblies are achieved by beginning another source module immediately following the END card from the previous source module. Commands can not be issued between two source programs. If any command is given, the assembler must be recalled.

## 2.3 ORGANISATION OF THE ASSEMBLER

The assembler is composed of ten modules which have the functions indicated:

| | |
|---|---|
| ICOMMON | Performs initialization, contains all private areas. |
| ICONTROL | Initializes the file tables for each pass, calls each pass of the assembler. |
| IOPACKGE | All I/O subroutines are contained in this module. |
| ISLU | Reads in source language and performs update processing. |
| IPASS1 | Reformats the input statement into compressed form, stores macro definitions, searches macro libraries. |
| IPASS2 | Expands macro calls. |
| IPASS3 | Determines the number of bytes of memory for the assembled program, builds a symbol table, formats DC constants. |
| IPAS3A | Relocates all symbols, builds the CSECT table, sorts the symbol table, relocates the LTORG table, outputs ESD information. |
| IPASS4 | Expands the operand field of instructions, outputs TXT, RLD, and END information, prints the assembly listing, outputs diagnostic file for ADIAG routine. |
| IPAS4A | Prints the cross-reference listing (XREF). |

The assembler can be run as a partially-shared program. It is composed of two parts: one part can only be run as nonshared, the other part can be run as either shared or nonshared.

The two modules ICOMMON and ISLU are linked into one single object module by the Linkage Editor. This object module can be run as nonshared only.

All remaining modules are loaded dynamically and are stored in the object module library: ASSEMB.LINK.S3020100.

The module ICONTROL can only be run nonshared.

The modules IPASS1, IPASS2, IPASS3, IPAS3A, IPASS4, IPAS4A can be run as either shared or nonshared.

When the command /EXEC ASSEMB is issued, the Static Loader loads the object module, which issues a LINK call to Dynamic Linking Loader to load the ICONTROL module from the OML file. V-type constants defining the entry points of the remaining 7 modules (IOPACKGE, IPASS1, IPASS2, IPASS3, IPAS3A, IPASS4, IPAS4A) are stored in the IOCONTROL module. When DLL loads the IOCONTROL module, the same OML file is searched and the remaining 7 modules are loaded at the same time as IOCONTROL in order to satisfy all the V-type constants stored in ICONTROL.

SHARE commands must be issued by the System Administrator in order to run the modules as shared.

Example 1: 6 modules are to be shared.

The System Administrator issues the following command:

```
/SHARE (IPASS1,IPASS2,IPASS3,IPAS3A,IPASS4,IPAS4A),
 ASSEMB.LINK.S3020100
```

Note

If there is not enough space to issue the entire SHARE command on the same line, several SHARE commands can be issued to achieve the same purpose.

Example 2: Not all 7 modules are shared

For modules IPASS2 and IPASS4 to be shared, the System Administrator has to issue the following command:

```
/SHARE (IPASS2,IPASS4),ASSEMB.LINK.S3020100
```

## 2.4 INPUT TO THE ASSEMBLER

The assembler assumes that the source program is in a SYSDTA file, and reads it using the RDATA macro. When the source correction facility is used, the assembler reads the source program from a tape file using BTAM.

The SYSDTA file may be one of the following:

1. A card file which is placed in a temporary system file on the disk.

2. A cataloged SAM file.

3. A cataloged ISAM file.

In addition the source program may be entered at a data display terminal, or read by the assembler from a program library created in accordance with LMS conventions (see section 6.1 COMOPT).

Note on 2 and 3:

It is not permitted to convert SAM files to ISAM files or vice versa during assembly.

**1. Sequential Files**

In put records from card reader files and SAM files will be treated as 80-character card images. The standard columns are 1 for start, 71 for end, and 16 for continuation.
Standard columns may be changed by the assembly ICTL statement.

BS2000 also accepts input records of up to 128 characters. However, only the first 80 bytes are processed, the rest is insignificant. Records shorter than 80 bytes will be space-filled (see Appendix A-9).

**2. Indexed Sequential Files**

Indexed sequential files will be treated as variable-length card images. Images which are less than 80 characters will be space-filled to the right, and images which are longer than 80 characters will be truncated. Since ISAM input files may be in File Editor format, an eight-byte key will be assumed to be at the beginning of each record. The assembler aligns the start column to character 9, the end column to character 79 and the continue column to character 24.

The eight-byte ISAM key is transferred by the assembler to the card number field (columns 73-80). The old card numbers of the input cards are overwritten in the process. The card number field in the assembly listing will no longer contain the old card numbers, but the corresponding ISAM keys.
There are, however, a few exceptions:

- The first statement in the source program is an ICTL statement in which a value higher than 71 is specified for the END column.

- Though the source program is stored as an ISAM file, the assembler reads corrections from a correction file. (See also Appendix A-8.)

- The source program is input via COMOPT SOURCE.

### 3. Terminal Input

Input from a terminal will be treated as variable-length card images. Records which are not equal to 80 bytes will be space-filled or truncated as in indexed sequential input. Columns 1, 71, and 16 will be used as the start, end, and continue columns. Column 1 means the first position at which the terminal I/O routine releases the keyboard to the user. Horizontal tab characters are not recognized by the assembler.

### 4. Library Input

Source programs and COPY members can also be read from program libraries generated according to LMS conventions.

### 5. Macro Libraries

A collection of macro definitions may be made available to more than one source program by placing the macros in a macro library. The macro definition may then be referenced by writing only the macro instruction call line.

Two kinds of macro libraries exist in the system: the system macro library (MACROLIB) which is available to all users, and the private macro libraries with the link names ALTLIB or ALTLIBn ($2 \leq n \leq 5$) which are cataloged as user files.

## 2.5 CONTROLLING THE ASSEMBLY SYSTEM

The assembly system can be controlled by options specified with the aid of the *COMOPT statement (see 6.1).
Control by the PARAM command is still supported (see 6.2).

## 2.6 OUTPUT FROM THE ASSEMBLER



### 1. Program Listings

The program listings produced are: a source correction listing, a listing of the External Symbol Dictionary (ESD), the source program with the corresponding object code and associated flags, the cross-reference listing, and diagnostic information. Each listing line is written by means of the WRLST Executive macro to a temporary disk file. The listing file is not saved after it has been spooled out to the printer.

### 2. Object Module File

The generated object module file is written either to the EAM file or to a program library created in accordance with LMS conventions.

### 3. Diagnostic File

The diagnostic file contains: the External Symbol Dictionary listing, a source program with corresponding object program listing, the cross-reference listing, error data, and a summary of errors. The diagnostic file is built sequentially as an ISAM file, so that it can be accessed by the Assembly Diagnostic Program nonsequentially.

### 4. Error File

The error file is a reduced diagnostic file in which only the invalid lines of the listing as well as error information are stored.

### 5. ERROR POOL

If the assembly is terminated when the abortion condition occurs (flag with error weight 3 or MNOTE with error code), the assembler outputs a special listing. This contains all lines so far determined as invalid together with the error descriptions, all MNOTES which have occured and any macro call during generation of which the abortion condition occurred.

Note

The scope of the information output to the ERROR POOL depends on the time of the abortion. If an abortion occurs in PASS1 due to overflow of a VM area, an additional message containing the name of the macro just reading is issued:

*** FATAL ERROR;PROCESSING MACRO xxxxx

## 2.7 CONSIDERATIONS FOR USE

### 2.7.1 Non-File Editor ISAM Source Files

Since ISAM input files are normally in File Editor format, an eight-byte key is assumed to be at the beginning of each record. If a user uses an ISAM input file which is not in File Editor format, then an ICTL assembly instruction must be used to set the standard columns as required by the file format.

Note, however, that when the assembler detects that SYSDTA is an ISAM file, it automatically adjusts column 1 to the ninth byte of the ISAM record. Therefore, any data prior to the ninth byte will be lost.

### 2.7.2 User Specification of Macro Libraries

The system macro library is specified with an FCB which has LINK=SYSLIB. If a user wishes to supply his own file as the system macro library, he may issue a /FILE command with LINK=SYSLIB for the new macro library or he may use the default filename, which is MACROLIB; the corresponding /RELEASE command must be issued by the user as well.

In addition, up to five private user macro libraries may be specified. Before the assembly run is started, the corresponding filenames and link names (ALTLIB, ALTLIB2, ALTLIB3, ALTLIB4 or ALTLIB5) must be specified in the /FILE command. Via *COMOPT control, the assembler is informed of the presence of the user macro libraries (in the case of only one private macro library this can also be achieved via the /PARAM command ALTLIB=YES).

Macros may also be contained as TYP=M members of a program library generated according to LMS conventions.
Assignments are likewise specified using ALTLIB.

If a program library contains several TYP=M members with identical names but different version numbers, the member having the highest version number is used.

### 2.7.3 Priority Relationship in Macro Libraries

Both the system macro library and up to five user macro libraries can be referenced in the same assembly. In the event of a macro the macro definitions are sought in accordance with the following rules:

- Each macro definition is read once only.
- The sequence of the macros in the source program is immaterial for the read algorithm.
- For each macro that is not an inner macro the macro definitions are sought and read in accordance with the following search hierarchy:

  Source program
  ALTLIB
  ALTLIB2
  ALTLIB3
  ALTLIB4
  ALTLIB5

  Subsequently, all first level inner macros are sought and, if this has not yet be done, read in; then all second level inner macros, etc.

  If the last macro level is closed and there are still any undefined macros, then a search for them is made in the system macro library (SYSLIB). Inner macros of macros from the SYSLIB are only sought in the SYSLIB if the corresponding macro definitions have not been read in previously.

For example, if a system macro, M1, calls on macros PR1 and PR2, then the user may include PR1 and PR2 in his own library and they may be referenced by the source program without being generated in the following manner:

| Name | Operation | Operand |
|---|---|---|
| | AGO | .SKIP |
| | PR1 | |
| | PR2 | |
| .SKIP | ANOP | |

Macros PR1 and PR2 will then be taken from the user's private library.

### 2.7.4 Obtaining an Object Module on Cards

The BS2000 assembler will not punch object cards. A user must issue a /PUNCH command for the Object Module File which is created by an assembly.

### 2.7.5 Obtaining an Assembly Listing from the Diagnostic File

When SAVLST=ALL is specified, production of the normal listing (with ASMLST=YES) is redundant. The listing can be produced from the diagnostic file by using the PRINT or END L ADIAG statements.

### 2.7.6 Diagnostic File and Error File Naming

The assembler forms the name of the diagnostic file by suffixing the characters SAVLST.ASSEMB. with the name of the program's first CSECT. Thus if the first CSECT (or the START statement) of a program is named ZZZ, the diagnostic file for that program has the name SAVLST.ASSEMB.ZZZ. If the first CSECT is unnamed, the file name will be SAVLST.ASSEMB.<TSN-number>. This means that two programs having the same CSECT name will also have the same diagnostic file name.
It may be necessary to change the name of the first program's diagnostic file to prevent it being overwritten. See the Data Management System Manual for the methods available to change an existing file name.

If the diagnostic file is to be given a different name, an additional FILE command with the LINK=SAVLINK option (e.g. /FILE DIAG,LINK=SAVLINK) has to be input before the assembler is invoked. During assembly, the diagnostic file will then get the file name that is specified in the FILE command i.e. DIAG in the present example) and the file link name SAVLINK.

Similarly, the name of the error file will get the format ERRFIL.ASSEMB.<1st SECT name>. The name can be chosen at random with the aid of the LINK=ERRLINK option.

### 2.7.7 Saving or Executing an Assembled Program

The object modules generated are either stored in a program library created in accordance with LMS conventions (see COMOPT MODULE) or in the EAM file.

The object modules contained in the temporary EAM file can be further processed as described below. On completion of the task the EAM file is deleted.

- Saving the object modules from EAM

  - in a program library with the aid of the LMS;
  - by means of an output on punched cards (/PUNCH␣*).

- Linking the object modules from EAM

  The TSOSLNK Linkage Editor links

  - either one or more object modules to form a load program, which it stores in a cataloged file,
  - or several object modules to form one object module, which it stores in the EAM file.

- Execution with the DLL

  The DLL is started by means of the /EXEC␣* command. The object modules are linked; the load program is loaded and started. The program is not saved.

# 3 ASSEMBLY LANGUAGE DESCRIPTION

## 3.1 ASSEMBLY LANGUAGE CODING CONVENTIONS



### 3.1.1 Statements

A source program is a sequence of source statements that are punched into cards. These statements may be written on the standard coding form, (Figure 3-1). One line of coding on the form is punched into one card. The vertical columns on the form correspond to card columns.

Space is provided on the form for program identification. This information is not punched into a card.

The body of the form (Figure 3-1) is composed of two fields: the statement field, columns 1-71, and the identification sequence field, columns 73-80. The identification sequence field is not part of a statement and is discussed following the subsection Comments Statements (3.1.1.4).

The entries (i.e., coding) composing a statement occupy columns 1-71 of a statement line and, of needed, columns 16-71 of successive continuation lines.

#### 3.1.1.1 Continuation Lines

When it is necessary to continue a statement on another line, the following rules apply:

1. Continuation may be specified by entering any nonblank character in column 72 of the statement line; if nothing is placed in column 72, the source statement is considered complete.

2. Continue the statement on the next line, starting in the continue column (16). Columns to the left of the continue column must be blank.

3. On comment cards, column 72 is ignored; there are no continuation cards for comments.

Two continuation lines are allowed for the operand field of an assembly or machine instruction. This does not apply to macro instructions and prototype statements in macro definitions (see section 4.7.4, Alternate Statement Form).

The comment field of any statement may be continued on one additional line.

### 3.1.1.2 Statement Boundaries

Source statements are normally contained in columns 1-71 of any statement lines and columns 16-71 of any continuation lines. Therefore, columns 1, 71, and 16 are referred to as the "begin", "end", and "continue" columns, respectively. This convention may be altered by use of the Input Format Control (ICTL) assembly instruction discussed later in this publication.

However, macro definitions must be written in standard form.

Siemens-Datenverarbeitung

Assembler

Programm

Bearbeiter

Datum

Problem Blatt

Name | Operation | Operanden und Bemerkungen

Spalten 73—78 siehe Kopfzeile

Mehrfachlochungen sind in derselben Zeile als Bemerkung unter Angabe der Lochkombination zu erläutern.
Die Spalte 80 ist nur bei Einschiebungen auszufüllen. Es sollte mit „A" begonnen werden. Einschübe können in die letzten sechs Zeilen geschrieben werden.

Bestell-Nr. D 150/350 280500 (0001 8)

Fig. 3-1 Standard Coding Form

### 3.1.1.3 Statement Format

There are two types of statement: instructions and comments.

**Instructions** may consist of one to four entries; a name entry, an operation entry, an operand entry, and a comments entry. These entries must be separated by one or more blanks, and must be written in the order stated above. Total statement size is limited only by the continuation line restriction and the rules governing the syntax of individual items used in the operand field.

The coding form (Figure 3-1) is ruled to provide an eight-character name field, a five-character operation field, and a 56-character operand and/or comments field.

If desired, the programmer may disregard these boundaries and write the name, operation, operand, and comment entries in other positions, subject to the following rules:

1. The entries must not extend beyond statement boundaries (either the conventional boundaries, or those fixed by the programmer via the ICTL instruction).

2. The entries must be in proper sequence, as stated above.

3. The entries must be separated by one or more blanks.

4. If used, a name entry must be written starting in the begin column.

5. The name and operation entries must be completed on the first line of the statement, including at least one blank following the operation entry.

A description of the name, operation, operand, and comments entries follows:

**Name Entries:** The name entry is a symbol, composed of eight characters, or less, created by the programmer to identify a statement. A name entry is usually optional, but, if present, must be entered with the first (or only) character appearing in the begin column. If the begin column is blank, the assembler assumes no name has been entered. Blanks must not appear within a name entry, whether the symbol was introduced directly by the programmer or indirectly by conditional assembly or macro generation.

**Operation Entries:** The operation entry is the mnemonic operation code specifying the desired machine operation, macro, or assembler function. An operation entry is mandatory and must appear in the first statement line, starting at least one position to the right of the begin column. Valid mnemonic operation codes for machine and assembler operations are contained in Appendices A-6 and A-7 of this publication. Valid operation codes consist of five characters or less for machine or assembler operation codes, and eight characters or less for macro-instruction operation codes. No blanks may appear within the operation entry.

The mnemonic operation code can be changed by means of the OPSYN statement (see 3.32.10).

**Operand Entries:** The operand entries contain the codes which describe or address the storage locations, masks, storage-area length or types of data to be processed.

Depending on the needs of the instruction, one or more operands may be written. Operands are required for all machine instructions.

Operands must be separated by commas. Blanks must not intervene between operands and the commas that separate them.

The operands may not contain embedded blanks except as follows:
If character representation is used to specify a constant, a literal, or immediate data in an operand, the character string may contain blanks, e.g. C'AP␣D'.

**Comments Entries:** Comments are descriptive items of information about the program that are to be inserted in the program listing. All 256 valid characters, including blanks, may be used in writing a comment. The entry cannot extend beyond the end column (normally column 71), and a blank must separate it from the operand.

In instructions where an operand entry is not present but a comments entry is desired, the absence of the operand entry must be indicated by a comma preceded and followed by one or more blanks, as follows:

| Name | Operation | Operand |
|---|---|---|
| | CSECT | , COMMENT |
| | . | |
| | . | |
| | . | |
| | . | |
| | END | , COMMENT |

Instruction Example:

The following example illustrates the use of name, operation, operand, and comments entries. A compare instruction has been named by the symbol COMP, the operation entry (CR) is the mnemonic operation code for a register-to-register compare operation, and the two operands (5,6) designate the two general registers whose contents are to be compared.

The comments entry reminds the programmer that with this instruction he is comparing "new" sum with "old".

| Name | Operation | Operand |
|---|---|---|
| COMPL | CR | 5,6 NEW SUM TO OLD |

Summary of Instruction Format:

- The entries in an instruction must always be separated by at least one blank and must be in the following order: name, operation, operand(s), comment.
- Every statement requires an operation entry. Name and comment entries are optional. Operand entries are required for all machine instructions and most assembly instructions.
- The name and operation entries must be completed in the first statement line, including at least one blank following the operation entry.
- The name and operation entries must not contain blanks. Operand entries must not have blanks preceding or following the commas that separate them.
- A name entry must always start in the "begin" column.
- If the column after the end column is blank, the next line must start a new statement. If the column after the end column is not blank, the following line will be treated as a continuation line.
- All entries must be contained within the designated begin, end, and continue column boundaries.

### 3.1.1.4 Comments Statements

Comments statements are used to include a programmer's notes on in assembly listing. (These notes can be helpful during debugging and maintenance of a program). Comments statements have no effect on the assembled program; they are only printed in the assembly listing and, therefore, may appear at any point. Extensive notes, or comments, may be written by using a series of comments statements.

There are two types of comments statements. One type, written with an asterisk (*) in the begin column, is used for comments on the source program. The other type, written with a period in the begin column and followed by an asterisk, is used for comments on a macro definition. This type is further described in chapter 4 entitled "How to Prepare Macro Definitions".

An example of the comments statement is:

| Name | Operation | Operand | Col. 72 or equivalent |
|---|---|---|---|
| * THIS COMMENT IS CONTINUED<br>* ON ANOTHER LINE. | | | X (is ignored) |

### 3.1.1.5 Identification-Sequence Field

The identification-sequence field of the coding form (columns 73-80) is used to enter program identification and/or statement sequence characters. The entry is optional. If the field, or a portion of it, is used for program identification, the identification is punched in the statement cards, and reproduced in the printed listing of the source program.

As an aid to keeping source statements in order, the programmer may code an ascending sequence of characters in this field or a portion of it. These characters are punched into their respective cards, and, during assembly, the programmer may request the assembler to verify this sequence by use of the Input Sequence Checking (ISEQ) assembly instruction. This instruction is discussed in section 3.33.2.

### 3.1.1.6 Character Set

Source statements are written using the following characters:

**Letters:** A through Z, and $, # and @

**Digits:** 0 through 9

**Special characters:** + - , = . * ( ) ' / & blank

These characters are represented by the card punch combination and internal bit configuration listed in Appendix A-3. In addition, any of the 256 punch combinations may be designated by characters that appear between paired apostrophes (hexadecimal representation in comments, and in macro instruction operands).

## 3.2 ASSEMBLY LANGUAGE STRUCTURE

The basic structure of the language can be stated as follows.

A source statement is composed of:

- A name entry (usually optional).
- An operation entry (mandatory).
- An operand entry (usually required).
- A comments entry (optional).

Notes

- A name entry is a symbol.
- An operation entry is a mnemonic operation code representing a machine, assembly or macro instruction.
- An operand entry is one or more operands composed of one or more expressions, which, in turn, are composed of a term or an arithmetic combination of terms.
- Operands of machine instructions generally represent such things as storage locations, general registers, immediate data, or constant values. Operands of assembly instructions provide the information needed by the assembler program to perform the designated operation.

Terms shown in Figure 3-2 are classed as absolute or relocatable. Terms are absolute or relocatable depending on the effect of the program relocation upon them. (Program relocation is the loading of the object program into storage locations other than those originally assigned by the assembler program.) A term is absolute if its value does not change upon relocation. A term is relocatable if its value changes upon relocation.

The following subsection, Terms and Expressions, discusses these items as outlined in Fig. 3-2.

## 3.3 TERMS AND EXPRESSIONS

### 3.3.1 Terms



Every terms represents a value. This value may be assigned by the assembler program (symbols, symbol length attributes, location counter reference) or may be inherent in the term itself (self-defining term, literal).

An expression, an arithmetic combination of terms, is reduced to a single value by the assembly program.

Each type of term and the rules for its use will now be discussed.

### 3.3.2 Symbols

A symbol is a character or combination of characters used to represent locations or arbitrary values. Symbols, through their use in name fields and in operands, provide the programmer with an efficient means of naming and referencing a program element. There are three types of symbols:

1. Ordinary symbols.
2. Variable symbols.
3. Sequence symbols.

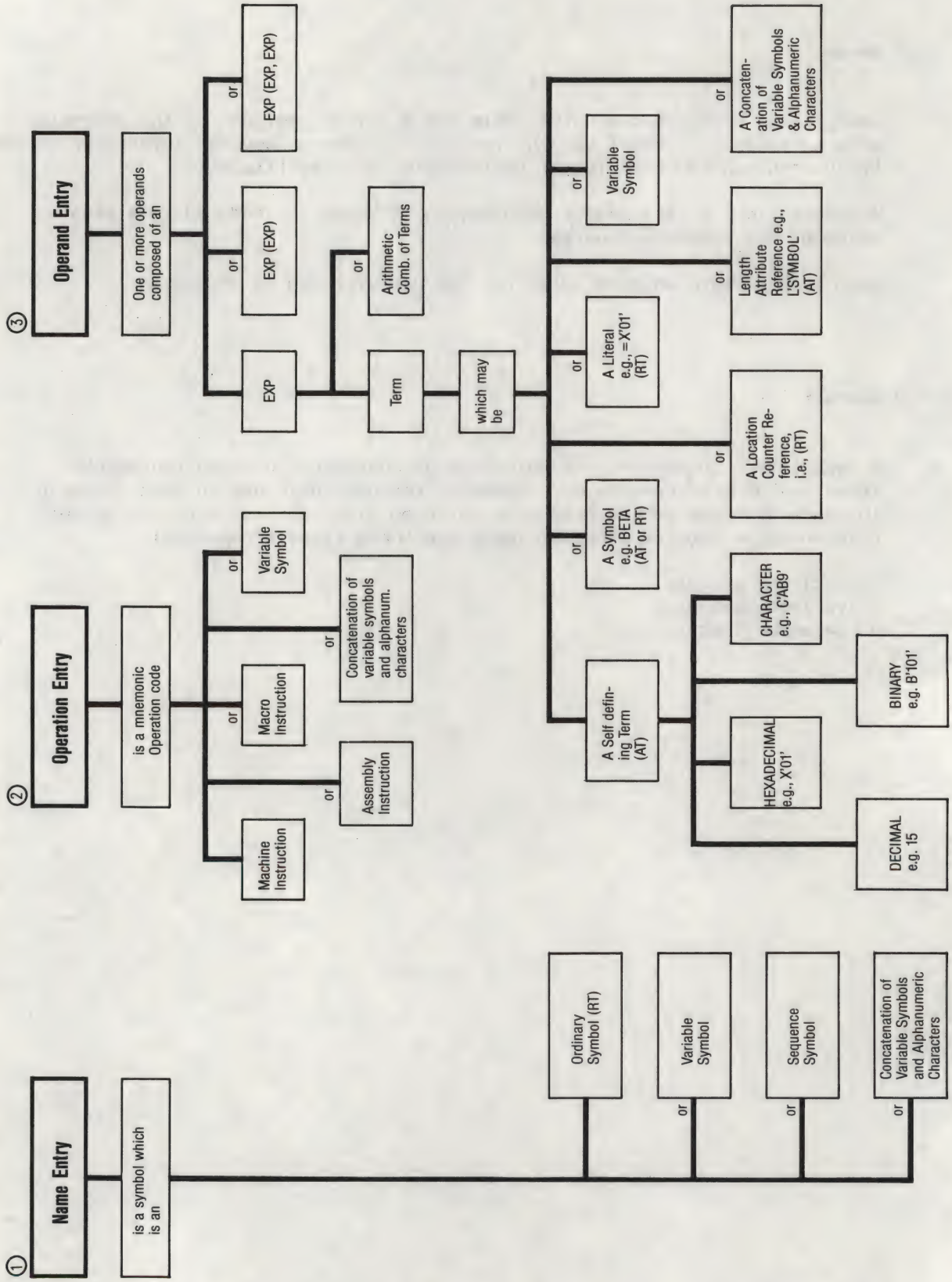



Fig. 3-2 Terms and Expressions

**Ordinary symbols** consist of one to eight letters and/or numbers, the first of which must be a letter. Such symbols are used to identify machine locations or arbitrary values. In the following sections, where symbols occur, they refer to this type of term.

**Absolute symbols** are ordinary symbols whose values do not change upon program relocation.

**Relocatable symbols** are ordinary symbols whose values change upon relocation.

The following are valid ordinary symbols:

READER
A23456
X4F2
LOOP2
N
S4
@B4
$A1
#56

The following ordinary symbols are invalid, for the reasons noted:

| | |
|---|---|
| 256B | First character is not alphabetic. |
| RECORDAREA2 | More than eight characters. |
| BCD*34 | Contains a special character - an asterisk. |
| IN AREA | Contains a blank. |

**Variable symbols** must begin with an ampersand (&) followed by one to seven letters and/or numbers, the first of which must be a letter. Variable symbols allow different values to be assigned to one symbol. The characters that replace a variable symbol must conform to the syntax rules governing the field in which the variable symbol occurs. (A complete discussion of variable symbols appears in a later section.)

**Sequence symbols** consist of a period (.) followed by one to seven letters and/or numbers, the first of which must be a letter. Sequence symbols are used to indicate the position of statements within the source program or macro definition. Through their use the programmer can vary the sequence in which statements are processed by the assembler program. Variable symbols cannot bs used to generate a sequence symbol in the name entry of a statement. (See the complete discussion in a later section.)

## 3.4 DEFINING SYMBOLS

The assembler assigns a value to each symbol appearing as a name entry in a source statement. The values assigned to symbols naming storage areas, instructions, constant, and control sections are the addresses of the leftmost bytes of the storage fields containing the named items. Since the addresses of these items may change upon program relocation, the symbols naming them are considered relocatable terms.

A symbol used as a name entry in the Equate Symbol (EQU) assembly instruction is assigned a value designated in the operand entry of the instruction. Since the operand entry may represent a relocatable value, or an absolute (i.e., nonchanging) value, the symbol is considered a relocatable term or an absolute term, depending on the value to which it is equated.

The value of a symbol may not be negative and may not exceed $2^{24}-1$.

A symbol is said to be defined when it appears as the name of a statement which appears in the source after all conditional assembly statements have been evaluated. (A special case of symbol definition is discussed in section 3.14, "Program Sectioning and Linking").

Symbol definition also involves the assignment of a length attribute to the symbol. (The assembler maintains an internal table - the symbol table - in which the values and attributes of symbols are kept. When the assembler encounters a symbol in an operand, it refers to the table for the values associated with the symbol). The length attribute of a symbol is the length, in bytes, of the storage field whose address is represented by the symbol. For example, a symbol naming an instruction that occupies four bytes of storage has a length attribute of 4. Note that there are exceptions to this rule: for example, in the case where symbol has been defined by an equate to location counter value (EQU*) or by a self-defining term, the length attribute of the symbols is 1. These and other exceptions are noted under the instructions involved. The length attribute is never affected by a duplication factor.

### 3.4.1 Previously Defined Symbols

The assembly language requires that symbols appearing in the operand entry of some instructions be previously defined. This simply means that the symbols, before their use in an operand, must have appeared as the name entry of a prior statement.

<u>For example:</u>

```
        .
        .
        .
SYM1    MVC     A,B
SYM2    EQU     SYM1
        .
        .
        .
```

would be a valid sequence of coding. The symbolic address of SYM1 must be defined before the EQU statement.

### 3.4.2 General Restrictions on Symbols

A symbol may be defined only once in an assembly. While the same symbol may appear as the name of two or more statements before macro generation and conditional assembly, only one such statement should be generated. In addition, a symbol may be used in the name field more than once as a control section name (i.e., defined in the START, CSECT, COM, or DSECT assembly statements) because the coding of a control section may be suspended and then resumed at any subsequent point. The CSECT or DSECT statement that resumes the section must be named by the same symbol that initially named the section; thus, the symbol that names the section must be repeated. Such usage is not considered to be duplication of a symbol definition.



## 3.5 SELF-DEFINING TERMS

A self-defining term is one whose value is inherent in the term. It is not assigned a value by the assembler program. For example, the decimal self-defining term >> 15 << represents a value of fifteen.

There are four types of self-defining terms: decimal, hexadecimal, binary, and character. Use of these terms is spoken of as decimal, hexadecimal, binary, or character representation of the machine language binary value or bit configuration they represent.

Self-defining terms are classed as absolute terms because the values they represent do not change upon program relocation.

### 3.5.1 Using Self-Defining Terms

Self-defining terms are the means of specifying machine values or bit configurations without equating the values to symbols and using the symbols. Self-defining terms may be used to specify such program elements as immediate data, masks, registers, addresses, and address increments.

The use of a self-defining term is quite distinct from the use of data constants or literals. When a self-defining terms is used in a machine-instruction statement, its value is assembled into the instruction. When a data constant or literal is specified in the operand of an instruction, its address is assembled into the instruction.

Note that variable symbols may be used to generate self-defining terms in assembly and machine instructions.

### 3.5.2 Decimal Self-Defining Term

A decimal self-defining term is simply an unsigned decimal number written as a sequence of decimal digits. High-order zeros may be used (e.g., 007). Limitations on the value of the term depend on its use. For example, a decimal term that designates a general register must have a value between 0 and 15 inclusive; one that represents an address must not exceed the size of storage. In any case, a decimal term may not consist of more than eight digits or exceed 16,777,215 ($2^{24}$-1). A decimal term is assembled as its binary equivalent. Some examples of decimal self-defining terms are 8147, 8147; 4092; 00021.

### 3.5.3 Hexadecimal Self-Defining Term

A hexadecimal self-defining term is an unsigned hexadecimal number written as a sequence of hexadecimal digits. The digits must be enclosed in single apostrophes and preceded by the letter X: X'C49'.

Each hexadecimal digit is assembled as its four-bit binary equivalent. Thus, a hexadecimal term used to represent an eight-bit mask would consist of two hexadecimal digits. The maximum value of a hexadecimal term is X'FFFFFFFF'.

The hexadecimal digits and their bit patterns are as follows:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | C | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | D | 1101 |
| 2 | 0010 | 6 | 0110 | A | 1010 | E | 1110 |
| 3 | 0011 | 7 | 0111 | B | 1011 | F | 1111 |

A table for converting from hexadecimal representation to decimal representation is provided in Appendix A-4.

### 3.5.4 Binary Self-Defining Term

A binary self-defining term is written as an unsigned sequence of 1's and 0's enclosed in apostrophes and preceded by the letter B, as follows: B'10001101'. This term would appear in storage as shown, occupying one byte. A binary term may have up to 32 bits represented. Padding with binary zero is on the left.

Binary representation is used primarily in designating bit patterns of masks or in logical operations.

The following example illustrates a binary term used as a mask in a Test Under Mask (TM) instruction. The contents of GAMMA are to be tested, bit by bit, against the pattern of bits represented by the binary term.

| Name | Operation | Operand |
|---|---|---|
| ALPHA | TM | GAMMA,B'10101101' |

### 3.5.5 Character Self-Defining Term

A character self-defining term consists of one to four characters enclosed by apostrophes. It must be preceded by the letter C. All letters, decimal digits, and special characters may be used in a character term. In addition, any of the remainder of the 256 punch combinations may be designated in a character self-defining term. Examples of character self-defining terms are as follows:

```
C'/'      C'  ' (two blanks)
C'ABC'    C'13'
```

Because of the use of apostrophes in the assembly language and ampersands in the macro language as syntactic characters, the following rule must be observed when using these characters in a character term. For each apostrophe or ampersand desired in a character term, two apostrophes or ampersands must be written. For example, the character value A'# would be written C'A''#', while an apostrophe followed by a blank and another apostrophe would be written as C''' ''''.

Each character in the character sequence is assembled as its eight-bit (EBCDIC) code equivalent (see Appendix A-3). The two apostrophes or ampersands that must be used to represent a single apostrophe or ampersand within the character sequence are assembled as a single apostrophe or ampersand.

## 3.6 LOCATION COUNTER REFERENCE

A Location Counter is used to assign addresses to program statements. It is the assembler program's equivalent of the instruction counter in the computer. As each machine instruction or data area is assembled, the Location Counter is first adjusted to the proper boundary for the item, if adjustment is necessary, and then incremented by the length of the assembled item. Thus, it always points to the next available location. If the statement is named by a symbol, the value assigned to the symbol is the value of the Location Counter after boundary adjustment, but before addition of the length.

The assembler maintains a Location Counter for each control section of the program and uses each Location Counter as previously described.

Source statements for each section are assigned addresses from the Location Counter for that section. The Location Counter for each successively declared control section assigns locations in consecutively higher areas of storage. If a program has multiple control sections, all statements identified as belonging to the first control section will be assigned from the Location Counter for section 1, the statements for the second control section will be assigned from the Location Counter for section 2, etc. This procedure is followed whether the statements from different control sections are interspersed or written in control section sequence.

The Location Counter setting can be controlled by using the START and ORG assembly instructions. The counter affected by either of these assembler instructions is the counter for the control section in which they appear. The maximum value for the Location Counter is $2^{24}-1$.

The programmer may refer to the current value of the Location Counter at any place in a program, by using an asterisk in an operand. The asterisk represents the location of the first byte of storage currently available (i.e., after any required boundary adjustment). Using an asterisk in a machine instruction statement is the same as placing a symbol in the name field of the statement and then using that symbol as an operand of the statement. Because a Location Counter is maintained for each control section, a Location Counter reference designates the Location Counter for the section in which the reference appears.

A reference to the Location Counter may be made in a literal address constant (i.e., the asterisk may be used in an address constant specified in literal form). The address of the instruction containing the literal is used for the value of the Location Counter. A Location Counter reference may not be used in a statement which requires the use of a predefined symbol, with the exception of the EQU and ORG assembly instructions.

## 3.7 LITERALS

A literal term is one of three basic ways to introduce data into a program. It is simply a constant preceded by an equal sign (=).

A literal represents data rather than a reference to the location of the respective data. The appearance of a literal in a source statement directs the assembler program to assemble the data specified by the literal, store this data in the "literal pool" of memory, and replace the literal in the statement by the address of the memory location containing the data.

Literals provide a means of entering constants (such as numbers for calculation, addresses, indexing factors, or words or phrases for printing out a message) into a program by specifying the constant in the operand of the instruction in which it is used. This is in contrast to using the DC assembly instruction to enter the data into the program, and then using the name of the DC instruction in the operand. Only one reference to a literal is allowed in a machine instruction statement.

A literal term may not be combined with any other terms.

A Literal may not be used as the receiving field of an instruction that modifies storage.

A literal may not be specified in an address constant (DC - Define Constant).

An S-type address constant may not be used as a literal.

The instruction coded below shows one use of a literal.

| Name | Operation | Operand |
|---|---|---|
| GAMMA | L | 10,=F'274' |

The statement GAMMA loads a register using a literal as the second operand. When assembled, the second operand of the instruction will be the address at which the binary value represented by F'274' is stored.

Further examples are included in the description of the DC instruction.

In general, literals may be used wherever a storage address is permitted as an operand. They may not, however, be used in any assembly instruction. Literals are considered relocatable, because the address of the literal, rather than the literal itself, will be assembled in the statement that employs a literal.

The assembler generates the literals, collects them, and places them in a specific area of storage, as explained in section 3.7.1, "The Literal Pool". A literal is not be confused with the immediate data in an SI instruction. Immediate date is assembled into the instruction.

### 3.7.1 Literal Format

The assembler requires as description of the type of literal being specified as well as the literal itself. This descriptive information assists the assembler in assembling the literal correctly. The descriptive portion of the literal must indicate the format in which the constant is to be assembled. It may also specify the length the constant is to occupy.

The method of describing and specifying a constant as a literal is nearly identical to the method of specifying it in the operand of a DC assembly instruction. The difference is that the literal must start with an equal sign (=), which indicates to the assembler that a literal follows. The type of literal designated in an instruction is not checked for correspondence with the operation code of the instruction.

Some examples of literals are:

| | |
|---|---|
| =A(BETA) | address constant literal. |
| =F'1234' | a fixed-point number with a length of four bytes. |
| =C'ABC' | a character literal. |

**The Literal Pool:** The literals processed by the assembler are collected and placed in a special area called the literal pool, and the location of the literal itself is assembled in the statement employing a literal. The positioning of the literal pool may be controlled by the programmer with the aid of the LTORG statement. Unless otherwise specified, the literal pool is placed at the end of the first control section.

The programmer may also specify that multiple literal pools be created. However, the sequence in which literals are ordered within the pool is controlled by the assembler, not by the programmer.

## 3.8 SYMBOL LENGTH ATTRIBUTE REFERENCE

The length attribute of a symbol may be used as a term by coding L' followed by the symbol, as in: L'BETA.

The length attribute of BETA will be substituted for the term. The following example illustrates the use of L' symbol in moving a character constant into either the high-order or low-order end of a storage field.

To clarify the example, the length attributes of A1 and B2 are given.

| Name | Operation | Operand |
|---|---|---|
| A1 | DS | CL8 |
| B2 | DC | CL2'AB' |
| HIORD | MVC | A1(L'B2),B2 |
| LOORD | MVC | A1+L'A1-L'B2(L'B2),B2 |

A1 names a storage field eight bytes in length and is assigned a length attribute of eight.

B2 names a character constant two bytes in length and is assigned a length attribute of two.

The statement named HIORD moves the contents of B2 into the leftmost two bytes of A1. The term L'B2 in parentheses provides the length specification required by the instruction. When the instruction is assembled, the length is placed in the proper field of the machine instruction.

The statement named LOORD moves the contents of B2 into the rightmost two bytes of A1. The combination of terms A1+L'A1-L'B2 results in the addition of the length of A1 to the beginning address of A1, and the substraction of the length of B2 from this value. The result is the address of the seventh byte in field A1. The constant represented by B2 is moved into A1 starting at this address. L'B2 in parentheses provides length specification as in HIORD.

Note:

The length attribute of * is equal to the length of the instruction in which it appears, except in an EQU * statement without length operand (see 3.26), where the length attribute is 1.

## 3.9 EXPRESSIONS

Expressions, which are used in coding operand entries for assembly language statements, are composed of either a single term or an arithmetic combination of terms (see Figure 3-2). Arithmetically combined terms, enclosed in parentheses, may be used in combination with terms outside the parentheses.

For example:

14+BETA-(GAMMA-LAMBDA)

When terms in parentheses occur in combination with other terms, like (GAMMA-LAMBDA) in the example, the parenthesized terms are reduced first to a single value. This value may be absolute or relocatable, depending on the combination of terms. This value is then used in reducing the rest of the combination to another single value.

Parenthesized terms may be included within another set of terms in parentheses.

For example:

A+B-(C+D-(E+F)+10)

This expression has two levels of parentheses. A level of parentheses is a left parenthesis and its matching right parenthesis. One level of parentheses surrounds E+F. The next higher level of parentheses surrounds C+D-(E+F)+10. The innermost set of terms in parentheses (the lowest level) is evaluated first.

The following are examples of valid expression:

```
*                          TEN/TWO
AREA1+X'2D'                BETA*10
*+32                       B'101'
N-25                       C'ABC'
FIELD+332                  29
FIELD                      L'FIELD
(EXIT-ENTRY+1)+GO          LAMBDA+GAMMA
=F'1234'
ALPHA-BETA/(10+AREA*L'FIELD)-100
A*(A*(A*(A+1)+3*(B-3)))
```

The rules for coding expressions are:

1. An expression may not start with an arithmetic operator other than '-'. Thus, -A+BETA is valid but +A+5 is not.

2. An expression may not contain two terms or two operators in succession.

3. An expression may not contain more than 6 levels of parentheses or 6 uncombined subexpressions at any time.

4. A multi-term expression may not contain a literal.

### 3.9.1 Evaluation of Expressions

A single term expression, e.g., 29,BETA,*, or L'SYMBOL, takes on the value of the term involved.

A multi-term expression, e.g., BETA+10, ENTRY-EXIT, 25*10+A/B,) is reduced to a single value, as follows:

1. Each term is given its value.

2. Arithmetic operations are performed left to right. Multiplication and division are done before addition and subtraction, e.g., A+B*C is evaluated as A+(B*C), not (A+B)*C. The computed result is the value of the expression.

3. Every expression is computed to 32 bits.

4. Divisions always yields an integer result; any fractional portion of the result is dropped. E.G., 1/2*10 yields a zero result, whereas 10*1/2 yields 5.

5. Division by zero is valid and yields a zero result.

Parenthesized expressions used in an expression are processed before the rest of the terms in the expression, e.g., in the expression A+BETA*(CON+10), the term CON+10 is evaluated first and the resulting value used in computing the final value of the expression.

Final values of expressions must be between $-2^{24}$ and $2^{24}-1$ (inclusive). Intermediate results may have a value between $-2^{31}$ and $2^{31}-1$ (inclusive).

### 3.9.2 Absolute and Relocatable Expressions

An expression is called absolute if its value is unaffected by program relocation.

An expression is called relocatable if its value changes upon program relocation.

The two types of expressions, absolute and relocatable, take on these characteristics from the term or terms composing them. This relationship will now be discussed.

**Absolute Expressions:** An absolute expression may be an absolute term or any arithmetic combination of absolute terms. An absolute term may be an absolute symbol, any of the self-defining terms, or the length attribute reference. As indicated in Figure 3-2, all arithmetic operations are permitted between absolute terms.

An absolute expression may contain relocatable terms (RT) - alone or in combination with absolute terms (AT) - under the following conditions:

1. There must be an even number of relocatable terms in the expression.

2. The relocatable terms must be paired. Each pair of terms must have the same relocatability attribute, i.e., they appear in the same control section in this assembly. Each pair must consist of terms with opposite signs. The paired terms do not have to be contiguous, e.g., RT+AT-RT.

3. No relocatable expression may enter into a multiply or divide operation. (RT-RT)*10 is, however, valid.

The pairing of relocatable terms (with opposite signs and the same relocatability attribute) cancels the effect of relocation. Therefore, the value represented by the paired terms remains constant, regardless of program relocation. For example, in the absolute expression A+X-Y, A is an absolute term, and X and Y are relocatable terms with the same relocatability attribute. If A equals 50, Y equals 25, and X equals 10, the value of the expression would be 35. If X and Y are relocated by a factor of 100 their values would then be 125 and 110. However, the expression would still evaluate as 35 (50-125+110)=35).

An absolute expression reduces to a single absolute value.

The following examples illustrate absolute expressions. A is an absolute term; Y and X are relocatable terms with the same relocatability attribute.

```
A-Y+X
A
A*A
X-Y+A
*-Y
```

<u>Note</u>

A reference to the Location Counter must be paired with another relocatable term from the same control section, i.e., with the same relocatability attribute.

**Relocatable Expressions:** A relocatable expression is one whose value would change by n if the program in which it appears is relocated n bytes away from its originally assigned area of storage.

A relocatable expression may be a relocatable term. A relocatable expression may contain relocatable terms - alone or in combination with absolute terms - under the following conditions:

1. There must be an odd number of relocatable terms.

2. All the relocatable terms but one must be paired. Pairing is described under "Absolute Expressions".

3. No relocatable term may enter into a multiply or divide operation.

A relocatable expression reduces to a single relocatable value. This value is the value of the odd relocatable term, adjusted by the values represented by the absolute terms and/or paired relocatable terms associated with it.

For example, in the expression W-X+W-10W, and X are relocatable terms with the same relocatability attribute. If initially W equals 10 and X equals 5, the value of the expression is 5. However, upon relocation this value will the value of the expression is 5. However, upon relocation this value will change. If a relocation factor of 100 is applied, the value of the expression is 105. Note that the value of the paired terms, W-X, remains constant at 5 regardless of relocation. Thus, the new value of the expression, 105, is the result of the value of the odd term (W) adjusted by the values of W-X and 10.

The following examples illustrate relocatable expressions. A is an absolute term, W and X are relocatable terms with the same relocatability attribute, Y is a relocatable term with a different relocatability attribute.

```
Y-32*A                          =F'1234'      (literal)
W-X+Y                           A*A+W-W+Y
* (reference to                 W-X+W
   Location Counter)            Y
W-X+*
```

## 3.10 ADDRESSING - PROGRAM SECTIONING AND LINKING

The 4004 and 7.000 addressing technique requires the use of a base register, which contains the base address, and a displacement, which is added to the contents of the base register. The programmer may specify a symbolic address and request the assembler to determine its storage address in terms of a base register and a displacement. The programmer may rely on the assembler to perform this service for him by indicating which general registers are available for assignment and what values the assembler may assume each contains. The programmer may use as many or as few registers for this purpose as he desires. The only requirements are that, at the point of reference, a register containing an address from the same control section is available, and that this address is less than or equal to the address of the item to which the reference is being made. The difference between the two addresses may not exceed 4095 bytes.

## 3.11 ADDRESSES - EXPLICIT AND IMPLIED

An address is composed of a displacement plus the contents of a base register. In the case of RX instructions, the contents of an index register are also used to derive the address. Care must be taken not to index across segment boundaries.

The programmer writes an explicit address by specifying the displacement and the base register number. In designating explicit addresses a base register may not be combined with a relocatable symbol.

He writes an implied address by specifying an absolute or relocatable address. The assembler has the facility to select a base register and compute a displacement, thereby generating an explicit address from an implied address provided that it has been informed

- what base registers are available to it and
- what each contains.

The programmer conveys this information to the assembler through the USING and DROP assembly instructions.

## 3.12 BASE REGISTER INSTRUCTIONS

The USING and DROP assembly instructions enable programmers to use expressions representing implied addresses as operands of machine instruction statements, leaving the assignment of base registers and the calculation of displacement to the assembler. In order to use symbols in the operand field of machine instruction statements, the programmer must

- indicate to the assembler, by means of a USING statement, that one or more general registers are available for use as base registers,
- specify, by means of the USING statement, what value each base register contains, and
- load each base register with the value he has specified for it.

A program must have at least one USING statement for each control section that is implicitly addressed.

Having the assembler determine base registers and displacements relieves the programmer of separating each address into a displacement value and a base address value. This feature of the assembler will eliminate a likely source of programming errors, thus reducing the time required to check out programs. To take advantage of this feature, the programmer uses the USING and DROP instructions described in this subsection. The principal discussion of this feature follows the description of both instructions.

### 3.12.1 USING - Use Base Address Register

The USING instruction indicates that one or more general registers are available for use as base registers. This instruction also states the base address values that the assembler may assume will be in the registers at object time. Note that a USING instruction does not load the registers specified. It is the programmer's responsibility to see that the specified base address values are placed into the registers. Suggested loading methods are described in section 3.12.2., "Programming with the USING instruction".

The typical form of the USING instruction statement is:

| Name | Operation | Operand |
|---|---|---|
| Not used | USING | From 2-17 expressions of the form v,r1,r2,r3,...,r16 |

Operand v must be an absolute or relocatable expression with a value ranging from -2↑24↑ to +2↑24↑-1. No literals are permitted. Operand v specifies a value that the assembler can use as a base address. The other operands must be absolute expressions. The operand r1 specifies the general register that can be assumed to contain the base address represented by operand v. Operands r2, r3, r4,... specify registers that can be assumed to contain v+4096, v+8192, v+12288,..., respectively. The values of the operands r1, r2, r3,...,r16 must be between 0 and 15. For example the statement:

| Name | Operation | Operand |
|---|---|---|
| | USING | *,12,13 |

tells the assembler it may assume that the current value of the Location Counter in general registers 12, incremented by 4096, will be in general register 13 at object time.

If the programmer changes the value in a base register currently being used, and wishes the assembler to compute displacement from this value, the assembler must be told the new value by means of another USING statement. In the following sequence the assembler first assumes that the value of ALPHA is in register 9. The second statement then causes the assembler to assume that ALPHA+1000 is the value in register 9.

| Name | Operation | Operand |
|---|---|---|
| | USING | ALPHA,9 |
| | . | |
| | . | |
| | . | |
| | USING | ALPHA+1000,9 |

If the programmer has to refer to the first 4096 bytes of virtual memory, he can use general register 0 as a base register subject to the following conditions:

1. The value of operand v must be either absolute or relocatable zero or simply relocatable, and

2. register 0 must be specified as operand r1.

The assembler assumes that register 0 contains zero. Therefore, regardless of the value of operand v, it calculates displacements as if operand v were absolute or relocatable zero. The assembler also assumes that subsequent registers specified in the same USING statement contain 4096, 8192, etc.

Note:

If register 0 is used as a base register, the program is not relocatable, despite the fact that operand v may be relocatable. The program can be made relocatable by:

1. Replacing register 0 in the USING statement.

2. Loading the new register with a relocatable value.

3. Reassembling the program.

### 3.12.2 DROP - Drop Base Register

The DROP instruction specifies a register that was previously available and may no longer be used as a base register.

The typical form of the DROP instruction statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| Not used | DROP | Up to 16 absolute expressions of the form r1,r2,r3,...,r16 or blank |

The expressions indicate general registers previously specified in a USING statement that are now unavailable for base addressing. The following statement, for example, prevents the assembler from using registers 7 and 11 for base addressing:

| Name | Operation | Operand |
|---|---|---|
| | DROP | 7,11 |

It is not necessary to use a DROP statement when the base address in a register is changed by a USING statement; nor are DROP statements needed at the end of the source program. If a register that is not specified in a USING statement is used as an operand of a DROP statement, then an error flag is set.

A register made unavailable by DROP instruction can be made available again by a subsequent USING instruction. DROP with blank operand field releases all base registers.

#### Programming with the USING instruction

The USING and DROP instructions may be used anywhere in a program, as often as needed, to indicate the general registers that are available for use as base registers and the base address values the assembler may assume each contains at execution time. Whenever an address is specified in a machine instruction statement, the assembler determines whether there is an available register containing a suitable base address. A register is considered available for a relocatable address if it was assigned a relocatable value that is in the same control section as the address. A register assigned an absolute value is available for addressing absolute locations only. In either case, the base address is considered suitable only if it is less than or equal to the address of the item to which the reference is made. The difference between the two addresses may not exceed 4095 bytes. In calculating the base register to be used, the assembler always uses the available register giving the smallest displacement. If there are two registers with the same value, the register with the higher number is used.

| Name | Operation | Operand |
|---|---|---|
| BEGIN | BALR | 2,0 |
| | USING | *,2 |
| FIRST | . | |
| | . | |
| LAST | . | |
| | END | BEGIN |

In the preceding sequence, the BALR instruction loads register 2 with the address of the first storage location immediately following. In this case, it is the address of the instruction named FIRST. The USING instruction indicates to the assembler that register 2 contains this location. When employing this method, the USING instruction must immediately follow the BALR instruction. No other USING or load instructions are required if the location named LAST is within 4095 bytes of FIRST.

If operand v of a USING statement specifies an absolute value, the assembler will use the associated base registers only for operands which represent absolute values. In the absence of a base register containing a suitable absolute value, an operand whose absolute value is less than 4096 will be assembled with general register 0 as base register and its value as displacement. Thus, a base register does not have to be explicitly stated for the operands of shift-type instructions, for example.

In Table 3-1, the BALR and LM instructions load registers 2-5. The USING instruction indicates to the assembler that these registers are available as base registers for addressing a maximum of 16,384 consecutive bytes of storage, beginning with the location named HERE. The number of addressable bytes may be increased or decreased by altering the number of registers designated by the USING and LM instructions and the number of address constants specified in the DC instruction.

| Name | Operation | Operand |
|---|---|---|
| BEGIN | BALR | 2,0 |
| | USING | HERE,2,3,4,5 |
| HERE | LM | 3,5,BASEADDR |
| | B | FIRST |
| BASEADDR | DC | A(HERE+4096,HERE+8192,HERE+12288) |
| FIRST | . | |
| | . | |
| | . | |
| LAST | . | |
| | END | BEGIN |

Table 3-1: Multiple Base Register Assignment

## 3.13 RELATIVE ADDRESSING

Relative Addressing is the technique of addressing instructions and data areas by designating their location in relation to the Location Counter or to some symbolic location. This type of addressing is always in bytes, never in bits, words, or instructions. Thus, the expression *+4 specifies an address that is four bytes greater than the current value of the Location Counter. In the sequence of instructions shown in the following example, the location of the CR machine instruction can be expressed in two ways, ALPHA+2 or BETA-4, because all of the mnemonics in the example are for 2-byte instructions in the RR format.

| Name | Operation | Operand |
|---|---|---|
| ALPHA | LR | 3,4 |
| | CR | 4,6 |
| | BCR | 1,14 |
| BETA | AR | 2,3 |

## 3.14 PROGRAM SECTIONING AND LINKING

It is often convenient, or necessary, to write a large program in sections. The sections may be assembled separately, then combined subsequently into one object program. The assembler provides facilities for creating multi-sectioned programs and symbolically linking separately assembled programs or program sections. The combined number of control sections and dummy sections, plus the number of unique symbols in EXTRN statements, V-type address constants, symbols in external dummy sections, and dummy registers, may not exceed 2500. The total number of control sections, dummy sections and common sections may not exceed 512. If the same symbol appears in a V-type address constant and in the name field of a CSECT or DSECT statement, it is counted as two symbols.

Sectioning a program is optional, and many programs can best be written without sectioning them. The programmer writing an unsectioned program need not be concerned with the subsequent discussion of program sections, which are called control sections. There is no need to employ the CSECT instruction, which is used to identify the control sections of a multisection program. Similarly, there is no need to be concerned with the discussion of symbolic linkages if the program neither requires a linkage to nor receives a linkage from another program. If it is desired to identify the program and/or specify a tentative starting location, it may be done by using the START instruction. It may also be desirable to employ the dummy section feature obtained by using the DSECT instruction.

Notes:

- Program sectioning and linking is closely related to the specification of base registers for each control section.

- Sectioning and linking examples are provided under the heading "Addressing External Control Sections" (3.19).

## 3.15 CONTROL SECTIONS

The concept of program sectioning is a consideration at coding time, assembly time, and load time. To the programmer, a program is a logical unit. If it is desired to divide it into sections called control sections, it is written in such a way that control passes properly from one section to another regardless of the relative physical position of the sections in storage. A control section is a block of coding that can be relocated, independently of other coding, at load time without altering or impairing the operating logic of the program. It is normally identified by the CSECT instruction. However, if it is desired to specify a tentative starting location, the START instruction may be used to identify the first control section.

To the assembler, there is no such thing as a program, there is an assembly, which consists of one or more control sections. An unsectioned program is treated as a single control section. To the Linkage Editor, there are no programs, only control sections that must be fashioned into an object program.

### 3.15.1 Control Dictionary

The output of the assembler consists of the assembled sections and a control dictionary. The control dictionary contains information which the Linkage Editor needs in order to complete cross-referencing between control sections, as it combines them into an object program. The Linkage Editor can take control sections from various assemblies and combine them properly with the help of the corresponding control dictionaries. Successful combination of separately assembled control sections depends on the techniques used to provide symbolic linkages between the control sections.

Regardless of the degree to which a program is sectioned, the programmer still knows that it will eventually be placed in virtual storage as it was symbolically described. It is impossible to predict the exact location of individual control sections, since the virtual storage location assignments may be modified by the Linkage Editor or the Loader, and their physical location may be constantly changing within the time sharing environment.

### 3.15.2 Control Section Location Assignment

Control section contents can be intermixed because the assembler provides a Location Counter for each control section. Control sections are assigned starting locations consecutively, in the same order as the control sections first occur in the program. Each control section subsequent to the first begins at the next available double-word boundary unless the PAGE attribute is specified (see also Attributes of Control Sections, 3.17).

## 3.16 FIRST CONTROL SECTION

The first control section of a program has the following special properties:

1. The initial value of its virtual storage location counter may be specified as an absolute value.

2. It normally contains the literals requested in the program, although their positioning can be altered. This is further explained under the discussion of the LTORG assembly instruction.



### 3.16.1 START - Start Assembly

The START instruction may be used to give a name to the first (or only) control section of a program. It may also be used to specify the initial value of the location counter for the first control section of the program.

The typical form of the START instruction statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| A symbol or blank | START | A self-defining term or blank; may be followed optionally by one or more attribute names (separated by a comma) |

If a symbol names the START instruction, the symbol is established as the name of the control section. If not, the control section is considered to be unnamed. All subsequent statements are assembled as part of that control section. This continues until a CSECT instruction identifying a different control section or a DSECT instruction is encountered.
A CSECT instruction named by the name symbol that names a START instruction is considered to identify the continuation of the control section first identified by the START. Similarly, an unnamed CSECT that occurs in a program initiated by an unnamed START is considered to identify the continuation of the unnamed control section.

The symbol in the name field is a valid symbol whose value represents the address of the first byte of the control section. It has a length of one.

The assembler uses the self-defining term specified by the operand as the initial value of the location counter of the program. This value should be divisible by eight. If the value is not divisible by eight, the location counter is set at the next double-word boundary. For example, either of the following statements:

| Name | Operation | Operand |
|---|---|---|
| PROG2<br>PROG2 | START<br>START | 2040<br>X'7F8' |

could be used to assign the name PROG2 to the first control section and to indicate an initial assembly location of 2040. If the first operand is omitted, the assembler sets the initial value of the location counter to zero. This operand may not be omitted if one or more attribute names follow (e.g. START 0,READ).

Note:

The START instruction may not be preceded by any type of assembly language statement that may either affect or depend upon the setting of the location counter.

### 3.16.2 CSECT - Identify Control Section

The CSECT assembly instruction identifies the beginning or the continuation of a control section.

The format of the CSECT assembly statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| A symbol or blank | CSECT | One or more attribute names or blank |

If a symbol names the CSECT instruction, the symbol is established as the name of the control section; otherwise the section is considered to be unnamed. All statements following the CSECT are assembled as part of that control section until a statement identifying a different control section is encountered (i.e., another CSECT or a DSECT instruction).

The symbol in the name field is a valid relocatable symbol whose value represents the address of the first byte of the control section. It has a length attribute of one. The attributes of the control sections are described in section 3.17.

Several CSECT statements with the same name may appear within a program. The first is considered to identify the beginning of the control section; the rest identify the resumption of the section. Thus, statements from different control sections may be interspersed with one another. They are properly assembled (assigned contiguous storage locations) as long as the statements from the various control sections are identified by the appropriate CSECT instructions.

### 3.16.3 Unnamed Control Section

If neither a named CSECT instruction nor START instruction appears at the beginning of the program, the assembler assumes that it is to assemble an unnamed control section as the first (or only) control section. There may be only one unnamed control section in a program. If one is initiated and is then followed by a named control section, any subsequent unnamed CSECT statements are considered to resume the unnamed control section. If it is desired to write a small program that is unsectioned, the program does not need to contain a CSECT instruction.



### 3.16.4 DSECT - Identify Dummy Section

A dummy section represents a control section that is assembled but is not part of the object program. A dummy section is a convenient means of describing the layout of an area of storage without actually reserving the storage. (It is assumed that the storage is reserved either by some part of this assembly or else by another assembly). The DSECT instruction identifies the beginning or resumption of a dummy section. More than one dummy section may be defined per assembly, but each must be named.

The typical form of the DSECT instruction is as follows:

| Name | Operation | Operand |
|---|---|---|
| A symbol | DSECT | Not used; should be blank |

The symbol in the name field is a valid relocatable symbol whose value represents the first byte of the section. It has a length attribute of one.

Program statements belonging to dummy sections may be interspersed throughout the program or may be written as a unit. In either case, the appropriate DSECT instruction should precede each set of statements. When multiple DSECT instructions with the same name are encountered, the first is considered to initiate the dummy section and the reset to continue it.

Symbols that name statements in a dummy section may be used in USING instructions. Therefore, they may be used in machine instructions and data definitions where implicit addressing is desired. An example illustrating the use of a dummy section appears subsequently under "Addressing Dummy Sections".

Note:

A symbol that names a statement in a dummy section may be used in an A-type address constant in a DSECT, only if it is paired with another symbol (with the opposite sign) from the same dummy section; i.e., only if the constant is made non-relocatable.

**Dummy Section Location Assignment:**

A Location Counter is used to determine the relative locations of named program elements in a dummy section. The Location Counter is always set to zero at the beginning of the dummy section, and the location values assigned to symbols that name statements in the dummy section are relative to the initial statement in the section.

**Addressing Dummy Sections:**

The programmer may wish to describe the format of an area whose storage location will not be determined until the program is executed. He can describe the format of the area in a dummy section, and he can use symbols defined in the dummy section as the operands of machine instructions. To effect references to the storage area, he does the following:

1. Provides a USING statement which specifies a general register available as a base register and a value defined in the DSECT that the assembler may assume the register contains when resolving symbolic references to the DSECT.

2. Ensures that the same register is loaded with the actual address of the storage area.

The values assigned to symbols defined in a dummy section are relative to the initial statement of the section.
Thus, all machine instructions which refer to names defined in the dummy section will, at execution time, refer to storage locations relative to the address loaded into the register.

An example is shown in the following coding. Assume that two independent assemblies (assembly 1 and assembly 2) have been loaded and are not to be executed as a single overall program. Assembly 1 is an input routine that places a record in a specified area of storage, places the address of the input area containing the record in general register 3, and branches to assembly 2. Assembly 2 processes the record. The coding shown in the example is from assembly 2.

The input area is described in assembly 2 by the DSECT control section named INAREA. Portions of the input area (i.e., record) that the programmer wishes to work with are named in DSECT control section as shown. The assembly instruction USING INAREA,3 designates general register 3 as the base register to be used in addressing the DSECT control section, and that general register 3 is assumed to contain the address of INAREA.

Assembly 1, during execution, loads the actual beginning address of the input area in general register 3. Because the symbols used in the DSECT section are defined relative to the initial statement in the section, the address values they represent, will, at the time of program execution, be the actual storage locations of the input area.

```
Name      Operation   Operand
ASMBLY2   CSECT
BEGIN     BALR        2,0
          USING       *,2
          .
          .
          .
          USING       INAREA,3
          CLI         INCODE,C'A'
          BE          ATYPE
          .
          .
          .
ATYPE     MVC         WORKA,INPUTA
          MVC         WORKB,INPUTB
          .
          .
          .
WORKA     DS          CL20
WORKB     DS          CL18
          .
          .
          .
INAREA    DSECT
INCODE    DS          CL1
INPUTA    DS          CL20
INPUTB    DS          CL18
          .
          .
          .
          END
```

### 3.16.5 COM - Define Blank Common Control Section

The COM assembly instruction identifies and reserves a common area of storage that may be referred to by independent assemblies that have been linked and loaded for execution as one overall program.

More than one COM statement may appear within a program (see the previous discussion of PROGRAM SECTIONING for restrictions). A COM statement with the same name as an earlier one indicates the resumption of this section. Blank common is interpreted as having a name consisting entirely of spaces. Thus, only one blank common may be designated in an assembly.

When several assemblies are loaded, all of which have common control sections, the amount of storage reserved is equal to the longest common control section.

The format of the COM statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| A symbol or blank | COM | One or more attribute names or blank |

The common area may be broken up into subfields through the use of the DS and DC assembly instructions. Names of subfields are defined relative to the beginning of the common section, as in the DSECT control section.

No instructions or constants appearing in a common control section are assembled. Data can only be placed in a common control section through execution of the program.

If the assignment of common storage is done in the same manner by each independent assembly, reference to a location in common by any assembly results in the same location being referenced.

When assembled, common location assignment starts on the next available double word (or page) boundary following the last CSECT.

A common control section may be assigned attributes in the same manner as a CSECT.

### 3.16.6 XDSEC - Identify External Dummy Section

The XDSEC instruction has the following format:

| Name | Operation | Operand |
|---|---|---|
| NAME | XDSEC | D[EFINITION]/R[EFERENCE] |

The rules for the dummy section as described under 3.16.4 also apply to the external dummy section, with the following exceptions:

To define an external dummy section (XDSEC D) external information is transferred to the Linkage Editor, which uses this information to process the external information which is output as a reference to an external dummy section (XDSEC R) in another program. In the reference to an external dummy section (XDSEC R) the Location Counter is set to zero and remains zero throughout the whole section. The real Location Counter of a symbol in a reference to an external dummy section is entered by the Linkage Editor in the instruction using this symbol.

Thus, it is the Linkage Editor and not the assembler which evaluates the Location Counter for symbols in an XDSEC R. This has the advantage that modifying a symbol in an XDSEC D entails the reassembly of only those modules which use the symbol to be modified, and not of all the modules using this XDSEC.

EQU statements are not permitted in XDSEC.

Expressions in the operand field of an EQU statement outside of an XDSEC must not be a difference of XDSEC elements.

**Example of XDSEC**

```
SIEMENS F-ASSEMBLER LISTING

          SYMBOL   TYPE ID  ADDR LENGTH          EXTERNAL SYMBOL DICTIONARY

          PROG1    SD  0001 00000 0004C
          PROG2    VC  0002
          PROG3    VC  0003
  (DUMMY) EINGABE  XD  0004 00000 0001E
          VORNAME  XD  0005 00000 0000A
          NAME     XD  0006 0000A 00014

FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT

   000000                                  1    PROG1    CSECT
                                           2             PRINT NOGEN
   000000 05 20                            3             BALR  2,0
   000002                                  4             USING *,2
   000000                                  5             USING EINGABE,13
   000002 41 D0 2024     000026            6             LA    13,EININF
   000006 58 F0 2042     000044            7             L     15,VPROG2
   00000A 05 EF                            8             BALR  14,15
   00000C 58 F0 2046     000048            9             L     15,VPROG3
   000010 05 EF                           10             BALR  14,15
   000012                                 11             TERM
                                          13  2                *,VERSION 702
   000026 C1D5E3D6D5404040                22    EININF   DC    CL30'ANTON     MUELLER'
   000044 00000000                        23    VPROG2   DC    V(PROG2)
   000048 00000000                        24    VPROG3   DC    V(PROG3)
                                          25    *
   000000                                 26    EINGABE  XDSEC D
   000000                                 27    VORNAME  DS    CL10
   00000A                                 28    NAME     DS    CL20
                                          29             END
```

```
SIEMENS F-ASSEMBLER LISTING

          SYMBOL   TYPE ID  ADDR LENGTH          EXTERNAL SYMBOL DICTIONARY

          PROG2    SD  0001 00000 00012
  (DUMMY) EINGABE  XR  0002 00000 00000
          VORNAME  XR  0003 00000 0000A

FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT

                                          1    ********************************
000000                                    2    PROG2    CSECT
000000                                    3             USING *,15
000000                                    4             USING EINGABE,13
000000 D2 09 F008D000 000008 000000       5             MVC   AUSVN,VORNAME
000006 07 FE                              6             BR    14
000008                                    7    AUSVN    DS    CL10
                                          8    *
000000                                    9    EINGABE  XDSEC R
000000                                   10    VORNAME  DS    CL10
                                         11             END
```

```
SIEMENS F-ASSEMBLER LISTING

          SYMBOL   TYPE ID  ADDR LENGTH          EXTERNAL SYMBOL DICTIONARY

          PROG3    SD  0001 00000 0001C
  (DUMMY) EINGABE  XR  0002 00000 00000
          NAME     XR  0003 00000 00014

FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT

                                          1    ********************************
000000                                    2    PROG3    CSECT
000000                                    3             USING *,15
000000                                    4             USING EINGABE,13
000000 D2 13 F008D000 000008 000000       5             MVC   AUSNAHME,NAME
000006 07 FE                              6             BR    14
000008                                    7    AUSNAME  DS    CL20
                                          8    *
000000                                    9    EINGABE  XDSEC R
000000                                   10    NAME     DS    CL20
                                         11             END
```

PROG1 initializes the input information and defines the format of this information.
PROG2 processes the information VORNAME from the input information.
PROG3 processes the information NAME from the input information. The complete format of the input information is not known in PROG2 and PROG3, which only require the format of the used symbols from the input information.

## 3.17 ATTRIBUTES OF CONTROL SECTIONS

To facilitate dynamic linkage and loading, it is often necessary to indicate that certain attributes are characteristic of all data and instructions within a control section. The following attributes may be specified (in any order) in the operand field of CSECT and COM statements:

| | |
|---|---|
| PUBLIC | indicates that the section contains shared, public data or instructions. |
| READ | indicates that the section is never modified (read only). |
| PRVLGD | indicates that the section is to be assigned a protection key so that only privileged system service routines have read or write access to it. |
| PAGE | indicates that the section is to begin at a page boundary which is a multiple of 4096. |
| RESIDENT | indicates that the specified section is loaded in class-3 memory and held resident there. |

Attributes may be specified singly or in combination. The final set of attributes of a control section is determined by combining all attributes. The attributes may appear in the statement which defines the beginning of the control section and in any statements used to resume it. It is not necessary to repeat attributes for statements having identical names. However, for program documentation in the assembly listing it may be useful to repeat attributes in continuation sections. If no attributes are specified, the control section will be considered private, modifiable and double word oriented. Attribute information will be output for each control section as part of the object module.

## 3.18 SYMBOLIC LINKAGE

Symbols may be defined in one program and referred to in another, thus effecting symbolic linkages between independently assembled programs. The linkages can be effected only if the assembler is able to provide information about the linkage symbols to the Linkage Editor, which resolves these linkage references at load time. The assembler places the necessary information in the control dictionary, on the basis of the linkage symbols identified by the ENTRY and EXTRN instructions. Note that these symbolic linkages are described as linkages between independent assemblies; more specifically, they are linkages between independently assembled control sections.

In the program where the linkage symbol is defined (i.e., used as a name), it must also be identified to the Linkage Editor and assembler by means of the ENTRY assembly instruction. It is identified as a symbol that names an entry point, which means that another program may use that symbol in order to effect a branch operation or a data reference. The assembler places this information in the control dictionary.

Similarly, the program that uses a symbol defined in some other program must identify it by the EXTRN assembly instruction. Since the definition of the symbol appears in another program, the assembler arbitrarily assigns a length attribute of 1 and a value of 0. The assembler places this information in the control dictionary.

Another way to obtain symbolic linkages is by using the V-type address constant. The subsection "Data Definition Instructions" contains the details pertinent to writing a V-type address constant. It is sufficient here to note that this constant may be considered an indirect linkage point. It is created from an externally defined symbol, but that symbol does not have to be identified by an EXTRN statement. The V-type address constant is intended to be used for external branch references (i.e., for effecting branches to other programs). Therefore, it may not be used for external data references (i.e., for referring to data in other programs).

### 3.18.1 ENTRY - Identify Entry-Point Symbol

The ENTRY assembly instruction identifies linkage symbols that are defined in one program but which may be used by other programs.

The format of the ENTRY instruction statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| Not used | ENTRY | One or more relocatable symbols, separated by commas, that also appear as statement names |

The number of ENTRY symbols is not limited.

An ENTRY statement operand may not contain a symbol defined in a dummy section or blank common or unnamed CSECT. An ENTRY statement containing a symbol defined in an unnamed control section can be processed by the assembler, but the Linkage Editor will not process the resulting object module.

The following example identifies the statements named SINE and COSINE as entry points to the program.

| Name | Operation | Operand |
|---|---|---|
| | ENTRY | SINE,COSINE |

Notes:

- The name of a control section does not have to be identified by an ENTRY statement when another program uses it as an entry point. The assembler automatically places information on control section names in the control dictionary.

- Declaring a name as an ENTRY point does not constitute a definition for this name. The name will not be defined unless it appears in the name of an assembly or machine instruction. The length attribute of a symbol named in an ENTRY statement is therefore determined by the type of statement used to define it. The same symbol may not be used in the operand field of both an EXTRN and an ENTRY statement in the same assembly.

### 3.18.2 EXTRN - Identify External Symbol

The EXTRN assembly instruction identifies linkage symbols that are used by this program but defined in some other program. Each external symbol must be identified; this includes symbols that name control sections.

The typical form of the EXTRN instruction statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| Not used | EXTRN | One or more relocatable symbols, separated by commas. |

The symbols in the operand field may not appear as names of statements in this program. The following example identifies three external symbols that have been used as operands in this program but are defined in some other program.

| Name | Operation | Operand |
|---|---|---|
| | EXTRN<br>EXTRN | RATEBL, PAYCALC<br>WITHCALC |

A symbol declared as an EXTRN is considered defined after that point and may therefore be used wherever a storage address is desired. However, a separate USING statement is required for every external symbol used in this manner (note the second example in the following discussion). A symbol may be redundantly declared as an external symbol in more than one EXTRN statement. The length attribute of an external symbol is 1.

An example using the EXTRN instruction appears subsequently in section 3.19 under "Addressing External Control Sections".

Notes:

1. A V-type address constant must not be identified by means of EXTRN statements.

2. When external symbols are used in an expression, they may not be paired. Each external symbol must be considered as having a unique relocatability attribute.



### 3.18.3 WXTRN - Identify Conditional External Symbol

The WXTRN statement can be used instead of the EXTRN statement. When it is used, the automatic scanning of the module library for corresponding definitions by the Linkage Editor is suppressed. WXTRN symbols are resolved only if a corresponding definition is found by other control mechanisms during linking (e.g. by explicit linking with the INCLUDE statement).

The format of the WXTRN instruction statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| Not Used | WXTRN | One or more symbols separated by commas |

In all other respects the rules for the EXTRN statement apply.

## 3.19 ADDRESSING EXTERNAL CONTROL SECTIONS

A common way for a program to link to an external control section is to:

1. Create a V-type address constant with the name of the external symbol.

2. Load the constant into a general register and branch to the control section via the register.

The combined number of control sections and dummy sections plus the number of unique symbols in EXTRN statements, V-type address constants, symbols in external dummy sections, and dummy registers, must not exceed 2500. If the same symbol appears in a V-type address constant and in the name entry of a CSECT or DSECT statement, it is counted as two symbols.

| Name | Operation | Operand |
|---|---|---|
| MAINPROG | CSECT | |
| BEGIN | BALR | 2,0 |
| | USING | *,2 |
| | . | |
| | . | |
| | L | 3,VCON |
| | BALR | 1,3 |
| | . | |
| | . | |
| VCON | DC | V(SINE) |
| | END | |

For example, to link to the control section named SINE, the preceding coding might be used.

An external symbol naming data may be referred to as follows:

1. Identify the external symbol with the EXTRN instruction, and create an address constant from the symbol.

2. Load the constant into a general register, and use the register for base addressing.

For example, to use an area named RATETBL, which is in another control section, the following coding might be used:

| Name | Operation | Operand |
|---|---|---|
| MAINPROG | CSECT | |
| BEGIN | BALR | 2,0 |
| | USING | *,2 |
| | . | |
| | . | |
| | EXTRN | RATETBL |
| | . | |
| | . | |
| | L | 4,RATEADDR |
| | USING | RATETBL,4 |
| | A | 3,RATETBL |
| | . | |
| | . | |
| RATEADDR | DC | A(RATETBL) |
| | END | BEGIN |

## 3.20 MACHINE INSTRUCTION STATEMENTS

This section discusses the coding of the machine instructions represented in the assembly language. The reader is reminded that the functions of each machine instruction are discussed in the Processor Reference Manual.

Machine instructions may be represented symbolically as assembly language statements. The symbolic format of each varies according to the actual machine instruction format, of which there are five: RR, RX, RS, SI, and SS. Within each basic format, further variations are possible.

The symbolic format of a machine instruction is similar to, but does not duplicate, its actual format. A mnemonic operation code is written in the operation field, and one or more operands are written in the operand field. Comments may be appended to a machine instruction statement as explained under "Comments Entries".

Any machine instruction statement may be named by a symbol, which other assembly statements can use as an operand. The value attribute of the symbol is the address of the leftmost byte assigned to the assembled instruction. The length attribute of the symbol depends on the basic instruction format, as follows:

| Basic Format | Length Attribute |
|---|---|
| RR | 2 |
| RX | 4 |
| RS | 4 |
| SI | 4 |
| SS | 6 |

### 3.20.1 Instruction Alignment and Checking

All machine instructions are aligned automatically by the assembler on halfword boundaries. If any statement that causes information to be assembled requires alignment, the bytes skipped are filled with hexadecimal zeros. All expressions that specify storage addresses are checked to insure that they refer to appropriate boundaries for the instructions in which they are used. Register numbers are also checked to make sure that they specify the proper registers, as follows:

1. Floating-point instructions must specify floating-point registers 0, 2, 4 or 6.

2. Double-shift, full-word multiply, and divide instructions must specify an even-numbered general register in the first operand.

## 3.21 OPERAND FIELDS AND SUBFIELDS

Some symbolic operands are written as a single field and other operands are written as a field followed by one or two subfields. For example, addresses consist of the contents of a base register and a displacement. An operand that specifies a base and displacement is written as a displacement field followed by a base register subfield, as follows: 40(5). In the RX format, both an index register subfield and a base register subfield are written as follows: 40(3,5). In the SS format, both a length subfield and a base register subfield are written as follows: 40(21,5).

Table 3-2 shows two types of addressing formats for RX, RS, SI and SS instructions. In each case, the first type shows the method of specifying an address explicitly, as a base register and displacement. The second type indicates how to specify an implied address as an expression.

For example, a load multiple instruction (RS format) may have either of the following symbolic operands:

```
R1,R3,D2(B2)      explicit address
R1,R3,S2          implied address
```

Whereas D2 and B2 must be represented by absolute expressions, S2 may be represented either by a relocatable or an absolute expression.

In order to use implied addresses, the following rules must be observed:

1. The base register assembly instructions (USING and DROP) must be used.

2. An explicit base register designation must not accompany the implied address.

For example, assume that FIELD is a relocatable symbol, which has been assigned a value of 7400. Assume also that the assembler has been notified (by a USING instruction) that general register 12 currently contains a relocatable value of 4096 and is available as a base register. The following example shows a machine instruction statement as it would be written in assembly language and as it would be assembled. Not that the value of D2 is the difference between 7400 and 4096 and that X2 is assembled as zero, since it was omitted. The assembled instruction is presented in hexadecimal:

Assembly statement:

```
ST          4,FIELD
```

Assembled instruction:

```
OP Code     R1 X2 B2 D2
50          4  0  C  CE8
```

An address may be specified explicitly as a base register and displacement (and index register for RX instructions) by the formats shown in the first column of table 3-2. The address may be specified as an implied address by the formats shown in the second column. Observe that the two storage addresses required by the SS instructions are presented separately; an implied address may be used for one while an explicit address is used for the other.

| Type | Explicit Address | Implied Address |
|---|---|---|
| RX | D2(X2,B2) | S2(X2) |
| | D2(,B2) | S2 |
| RS | D2(B2) | S2 |
| SI | D1(B1) | S1 |
| SS | D1(L1,B1) | S1(L1) |
| | D1(L,B1) | S1(L) |
| | D2(L2,B2) | S2(L2) |

Table 3-2: Details of Address Specification

A comma must be written to separate operands. Parentheses must be written to enclose a subfield or subfields, and a comma must be written to separate two subfields within parentheses. When parentheses are used to enclose one subfield, and the subfield is omitted, the parentheses must be omitted. In the case of two subfields that are separated by a comma and enclosed in parentheses, the following rules apply:

1. If both subfields are omitted, the separating comma and the parentheses must also be omitted.

Example

```
L        2,48(4,5)
L        2,FIELD          (implied address)
```

2. If the first subfield in the sequence is omitted, the comma that separates it from the second subfield is written. The parentheses must also be written.

Example

```
MVC      32(16,5),FIELD2
MVC      32(,5),FIELD2    (implied length)
```

3. If the second subfield in the sequence is omitted, the comma that separates it from the first subfield must be omitted. The parentheses must be written.

Example

```
MVC      32(16,5),FIELD2
MVC      FIELD1(16),FIELD2   (implied length)
```

Fields and subfields in a symbolic operand may be represented either by absolute or by relocatable expressions, depending on what the field requires. (An expression has been defined as consisting of one term or a series of arithmetically combined terms). Refer to Tables 3-2 and 3-3 for a detailed description of field requirements.

Note:

Blanks may not appear in an operand unless provided by a character self-defining term or a character literal. Thus, blanks may not intervene between fields and the comma separators, between parentheses and fields, etc.

## 3.22 LENGTHS - EXPLICIT AND IMPLIED

The length field in SS instructions can be explicit or implied. To imply a length, the programmer omits a length field from the operand. The omission indicates that the length field is either of the following:

1. The length attribute of the expression specifying the displacement, if an explicit base and displacement have been written.

2. The length attribute of the expression specifying the effective address, if the base and displacement have been implied.

In either case, the length attribute for an expression is the length of the leftmost term in the expression. The length attribute of asterisk (*) is equal to the length of the instruction in which it appears, except in an EQU statement (see 3.26).

By contrast, an explicit length is written by the programmer in the operand as an absolute expression. The explicit length overrides any implied length.

Whether the length is explicit or implied, it is always an effective length. The value inserted into the length field of the assembled instruction is one less than the effective length in the source program statement.

Note:

If a length of zero is desired, the length may be stated as zero or one.

To summarize, the length required in an SS instruction may be specified explicitly by the formats shown in the first column of Table 3-3 or may be implied by the formats shown in the second column. Observe that the two lengths required in some of the SS instruction formats are presented separately. An implied length may be used for one while an explicit length is used for the other.

| Explicit Length | Implied Length |
|---|---|
| D1(L1,B1) | D1(,B1) |
| S1(L1) | S1 |
| D1(L,B1) | D1(,B1) |
| S1(L) | S1 |
| D1(L2,B2) | D1(,B2) |
| S2(L2) | S2 |

Table 3-3: Details of Length Specification in SS Instructions

## 3.23 MACHINE INSTRUCTION MNEMONIC CODES

The mnemonic operation codes (shown in Appendices A-6 and A-7) are designed so that their functions can be easily remembered. The normal format of the code is shown below; the items in brackets are not necessarily present in all codes:

Verb [Modifier][Data Type][Machine Format]

The verb, which is usually one or two characters, specifies the function. For example, A represents Add, and MV represents Move. The function may be further defined by a modifier. For example, the modifier L indicates a logical function, as in AL for Add Logical, and MV is modified by C (MVC) to indicate Move Characters.

Mnemonic codes for functions involving data usually indicate the data types, by letters that correspond to those for the data types in the DC assembly instruction.

Furthermore, letters U and W have been added to indicate short and long, unnormalized floating-point operations, respectively. For example, AE indicates Add Normalized Short, whereas AU indicates Add Unnormalized Short. Where applicable, full-word fixed-point data is implied if the data type is omitted.

The letters R and I are added to the codes to indicate, respectively, RR and SI machine instruction formats. Thus, AER indicates Add Normalized Short in the RR format. Functions involving character and decimal data types imply the SS format.

## 3.24 MACHINE INSTRUCTION EXAMPLES

The examples that follow are grouped according to machine instruction format. They illustrate the various symbolic operand formats. All symbols employed in the examples must be assumed to be defined elsewhere in the same assembly. All symbols that specify register numbers and lengths must be assumed to be equated elsewhere to absolute values (EQU instruction).

Implied addressing, control section addressing, and the function of the USING assembly instruction are not considered here. For discussion of these considerations and for examples of coding sequences that illustrate them, refer to section 3.14, Program Sectioning and Linking and section 3.12, Base Register Instruction.

**RR-Format:**

| Name | Operation | Operand |
|---|---|---|
| ALPHA1 | LR | 1,2 |
| ALPHA2 | LR | REG1,REG2 |
| BETA | SPM | 15 |
| GAMMA1 | SVC | 250 |
| GAMMA2 | SVC | TEN |

The operands of ALPHA1, BETA, and GAMMA1 are decimal self-defining values, which are categorized as absolute expressions. The operands of ALPHA2 and GAMMA2 are symbols that are equated elsewhere to absolute values.

**RX-Format:**

| Name | Operation | Operand |
|---|---|---|
| ALPHA1 | L | 1,39(4,10) |
| ALPHA2 | L | REG1,39(4,TEN) |
| BETA1 | L | 2,ZETA(4) |
| BETA2 | L | REG2,ZETA(REG4) |
| GAMMA1 | L | 2,ZETA |
| GAMMA2 | L | REG2,ZETA |
| GAMMA3 | L | 2,=F'1000' |
| LAMBDA1 | L | 3,20(,5) |

Both ALPHA instructions specify explicit addresses; REG1 and TEN are absolute symbols. Both BETA instructions specify implied addresses, and both use index registers. Indexing is omitted from the GAMMA instructions. GAMMA1 and GAMMA2 specify implied addresses. The second operand of GAMMA3 is a literal. LAMBDA1 specifies no indexing.

**RS-Format:**

| Name | Operation | Operand |
|---|---|---|
| ALPHA1 | BXH | 1,2,20(14) |
| ALPHA2 | BXH | REG1,REG2,20(REGD) |
| ALPHA3 | BXH | REG1,REG2,ZETA |
| ALPHA4 | SLL | REG2,15 |
| ALPHA5 | SLL | REG2,0(15) |

Whereas ALPHA1 and ALPHA2 specify explicit addresses, ALPHA3 specifies an implied address. ALPHA4 is a shift instruction shifting the contents of REG2 left 15 bit positions. ALPHA5 is a shift instruction shifting the contents of REG2 left by the value contained in general register 15.

**SI-Format**

| Name | Operation | Operand |
|---|---|---|
| ALPHA1 | CLI | 40(9),X'40' |
| ALPHA2 | CLI | 40(REG9),TEN |
| BETA1 | CLI | ZETA,TEN |
| BETA2 | CLI | ZETA,C'A' |
| GAMMA1 | SDV | 40(9) |
| GAMMA2 | SDV | 0(9) |
| GAMMA3 | SDV | 40(0) |
| GAMMA4 | SDV | ZETA |

The ALPHA instructions and GAMMA1-GAMMA3 specify explicit addresses, whereas the BETA instructions and GAMMA4 specify implied addresses. GAMMA2 specifies a displacement of zero. GAMMA3 does not specify a base register.

**SS-Format:**

| Name | Operation | Operand |
|---|---|---|
| ALPHA1 | AP | 40(9,8),30(6,7) |
| ALPHA2 | AP | 40(NINE,REG8),30(L6,7) |
| ALPHA3 | AP | FIELD2,FIELD1 |
| ALPHA4 | AP | FIELD2(9),FIELD1(6) |
| BETA | AP | FIELD2(9),FIELD1 |
| GAMMA1 | MVC | 40(9,8),30(7) |
| GAMMA2 | MVC | 40(NINE,REG8),DEC |
| GAMMA3 | MVC | FIELD2,FIELD1 |
| GAMMA4 | MVC | FIELD2(9),FIELD1 |

ALPHA1, ALPHA2, GAMMA1 and GAMMA2 specify explicit lengths and addresses. ALPHA3 and GAMMA3 specify both implied length and implied addresses. ALPHA4 and GAMMA4 specify explicit lengths and implied addresses. BETA specifies an explicit length for FIELD2 and an implied length for FIELD1; both addresses are implied.

### 3.24.1 Extended Mnemonic Codes

For the convenience of the programmer, the assembler provides extended mnemonic codes, which allow conditional branches to be specified mnemonically as well as through the use of the BC machine instruction. These extended mnemonic codes specify both the machine branch instruction and the condition on which the branch is to occur. The codes are not part of the universal set of machine instructions, but are translated by the assembler into the corresponding operation and condition combinations. The permissible extended mnemonic codes and their operand formats are shown in Table 3-4, together with their machine instructions in the RX format. Note that the only difference between the operand fields of the extended mnemonics and those of their machine instruction equivalents is the absence of the R1 field and the comma that separates it from the rest of the operand field. The extended mnemonic list, like the machine instruction list, shows explicit address formats only. Each address can also be specified as an implied address.

**Unconditional Branch Instructions**

| Extended Code | | Meaning | Machine Instruction |
|---|---|---|---|
| B | D2(X2,B2) | Branch Unconditional | BC 15,D2(X2,B2) |
| BR | R2 | Branch unconditional (RR-Format) | BCR 15,R2 |
| NOP | D2(X2,B2) | No operation | BC 0,D2(X2,B2) |
| NOPR | R2 | No Operation (RR-Format) | BCR 0,R2 |

**Used After Compare Instructions**

| Extended Code | | Meaning | Machine Instruction |
|---|---|---|---|
| BH | D2(X2,B2) | Branch on High | BC 2,D2(X2,B2) |
| BL | D2(X2,B2) | Branch on Low | BC 4,D2(X2,B2) |
| BE | D2(X2,B2) | Branch on Equal | BC 8,D2(X2,B2) |
| BNH | D2(X2,B2) | Branch on Not High | BC 13,D2(X2,B2) |
| BNL | D2(X2,B2) | Branch on Not Low | BC 11,D2(X2,B2) |
| BNE | D2(X2,B2) | Branch on Not Equal | BC 7,D2(X2,B2) |
| BRH | R2 | Branch on High (RR format) | BCR 2,R2 |
| BRL | R2 | Branch on Low (RR format) | BCR 4,R2 |
| BRE | R2 | Branch on Equal (RR format) | BCR 8,R2 |
| BRNH | R2 | Branch on Not High (RR format) | BCR 13,R2 |
| BRNL | R2 | Branch on Not Low (RR format) | BCR 11,R2 |
| BRNE | R2 | Branch on Not Equal (RR format) | BCR 7,R2 |

**Used After Arithmetic Instructions**

| Extended Code | | Meaning | Machine Instruction |
|---|---|---|---|
| BO | D2(X2,B2) | Branch on Overflow | BC 1,D2(X2,B2) |
| BP | D2(X2,B2) | Branch on Plus | BC 2,D2(X2,B2) |
| BM | D2(X2,B2) | Branch on Minus | BC 4,D2(X2,B2) |
| BZ | D2(X2,B2) | Branch on Zero | BC 8,D2(X2,B2) |
| BNP | D2(X2,B2) | Branch on Not Plus | BC 13,D2(X2,B2) |
| BNM | D2(X2,B2) | Branch on Not Minus | BC 11,D2(X2,B2) |
| BNZ | D2(X2,B2) | Branch on Not Zero | BC 7,D2(X2,B2) |
| BRO | R2 | Branch on Overflow (RR format) | BCR 1,R2 |
| BRP | R2 | Branch on Plus (RR format) | BCR 2,R2 |
| BRM | R2 | Branch on Minus (RR format) | BCR 4,R2 |
| BRZ | R2 | Branch on Zero (RR format) | BCR 8,R2 |
| BRNP | R2 | Branch on Not Plus (RR format) | BCR 13,R2 |
| BRNM | R2 | Branch on Not Minus (RR format) | BCR 11,R2 |
| BRNZ | R2 | Branch on Not Equal (RR format) | BCR 7,R2 |

**Used After Test Under Mask Instructions**

| Extended Code | | Meaning | Machine Instruction |
|---|---|---|---|
| BO | D2(X2,B2) | Branch on Ones | BC 1,D2(X2,B2) |
| BM | D2(X2,B2) | Branch on Mixed | BC 4,D2(X2,B2) |
| BZ | D2(X2,B2) | Branch on Zeros | BC 8,D2(X2,B2) |
| BNO | D2(X2,B2) | Branch on Not Ones | BC 14,D2(X2,B2) |
| BNM | D2(X2,B2) | Branch on Not Mixed | BC 11,D2(X2,B2) |
| BNZ | D2(X2,B2) | Branch on Not Zero | BC 7,D2(X2,B2) |
| BRO | R2 | Branch on Ones | BCR 1,R2 |
| BRM | R2 | Branch on Mixed | BCR 4,R2 |
| BRZ | R2 | Branch on Zeros | BCR 8,R2 |
| BRNO | R2 | Branch on Not Ones | BCR 14,R2 |
| BRNM | R2 | Branch on Not Mixed | BCR 11,R2 |
| BRNZ | R2 | Branch on Not Zeros | BCR 7,R2 |

Table 3-4: Extended Mnemonic Codes

In the following example, which illustrates the use of extended mnemonics, it is to be assumed that the symbol GO is defined elsewhere in the program.

| Name | Operation | Operand |
|---|---|---|
| | B | 40(3,6) |
| | B | 40(,6) |
| | BL | GO(3) |
| | BL | GO |
| | BR | 4 |

The first two instructions specify an unconditional branch to an explicit address. The address in the first case is the sum of the contents of base register 6, the contents of index register 3, and the displacement 40; the address in the second instruction is not indexed. The third instruction specifies a branch on low to the address implied by GO as indexed by the contents of index register 3; the fourth instruction does not specify an index register. The last instruction is an unconditional branch to the address contained in register 4.

## 3.25 ASSEMBLY INSTRUCTION STATEMENTS

Just as machine instructions are used to request the computer to perform a sequence of operations during program execution time, so assembly instructions are requests to the assembler to perform certain operations during the assembly. Assembly instruction statements, in contrast to machine instruction statements, do not always cause machine instructions to be included in the assembled program. Some, such as DS and DC, generate no instructions but do cause storage areas to be set aside for constants and other data. Others, such as EQU and SPACE, are effective only at assembly time; they generate nothing in the assembled program and have no effect on the Location Counter.

The following is a list of all the assembly instructions.

**Symbol Definition Instruction**

| | |
|---|---|
| EQU | Equate Symbol |

**Data Definition Instructions**

| | |
|---|---|
| DC | Define Constant |
| DS | Define Storage |
| DXD | Define Dummy Register |
| CXD | Define Length of Dummy Register Vector |

**Program Sectioning and Linking Instructions**

| | |
|---|---|
| START | Start Assembly |
| CSECT | Identify Control Section |
| DSECT | Identify Dummy Section |
| XDSEC | Identify External Dummy Section |
| ENTRY | Identify Entry-Point Symbol |
| EXTRN | Identify External Symbol |
| WXTRN | Identify Conditional External Symbol |
| COM | Identify Blank Common Section |

**Base Register Instructions**

| | |
|---|---|
| USING | Use Base Address Register |
| DROP | Drop Base Address Register |

### Listing Control Instructions

| | |
|---|---|
| TITLE | Identify Assembly Output |
| EJECT | Start New Page |
| SPACE | Space Listing |
| PRINT | Print Optional Data |

### Program Control Instructions

| | |
|---|---|
| ICTL | Input Format Control |
| ISEQ | Input Sequence Checking |
| ORG | Set Location Counter |
| LTORG | Begin Literal Pool |
| CNOP | Conditional No Operation |
| END | End Assembly |
| PUNCH | Output in Card Format |
| REPRO | Reproduce Following Card |

### Other Instructions

| | |
|---|---|
| COPY | Copy Library Member |
| OPSYN | Modify Mnemonic Code |
| STACK | Store PRINT, USING status |
| UNSTK | Reset PRINT, USING status |

## 3.26 SYMBOL DEFINITION INSTRUCTION

### 3.26.1 EQU - Equate Symbol



The EQU instruction is used to define a symbol by assigning to it the length, and relocatability attributes of an expression in the operand field.

The typical form of the EQU instruction statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| A symbol | EQU | An expression |

The expression in the operand field may be absolute or relocatable. Any symbols appearing in the expression must be previously defined.

The symbol in the name field is given the same attributes as the expression in the operand field. The length attribute of the symbol is that of the leftmost symbol of a multi-term expression in the operand field. However, if the expression in the operand field consists of a * or a self-defining term, the length attribute is 1. The value attribute of the symbol is the value of the expression. The value of the expression must not be more than 3 bytes.

The EQU instruction is the means of equating symbols to register numbers, immediate data, and other arbitrary values. The following examples illustrate how this might be done:

| Name | Operation | Operand |
|---|---|---|
| REG2 | EQU | 2 (general register) |
| TEST | EQU | X'3F' (immediate data) |

EQU statements have not yet been evaluated by the assembler when macro processing is being performed.

E.g.

| Name | Operation | Operand |
|---|---|---|
| RG1 | EQU | 1 |
| | MAC | RG1 |

causes the macro MAC to be generated, with the character sequence RG1 as operand, but not with 1.

To reduce programming time, the programmer can equate symbols to frequently used expressions and then use the symbols as operands in place of the expressions.

Thus, in the statement

| Name | Operation | Operand |
|---|---|---|
| FIELD | EQU | ALPHA-BETA+GAMMA |

FIELD is defined as ALPHA-BETA+GAMMA and may be used in place of it. Note that ALPHA, BETA and GAMMA must all be previously defined.

### 3.26.2 EQU - Extended Format

Two further expressions are permitted in the operand field, the first of which specifies the length attribute of the symbol defined and the second one its type. Both operands are optional; if they are omitted, the attributes are defaulted according to 3.26.1. When specifying the type attribute while omitting the length attribute, the two existing expressions must be separated by two commas.

The extended EQU format is this:

| Name | Operation | Operand |
|---|---|---|
| Name | EQU | expression1,[expression2],[expression3] |

The length attribute must be an absolute value in the range 0 to $2^{24}-1$. It may be analyzed both at macro language and at actual assembly time. To be analyzed as early as at macro processing time, however, the length attribute must be a simple decimal self-defining term.

Notes:

- When an expression or symbolic parameter is entered for the length, the standard value 1 is assigned at macro processing time; thus, different values may be assigned during the course of assembly.

- The type attribute can be evaluated only at macro language time. Therefore, it, too, must be specified as a simple self-defining value in the range 0-255, in character, hexadecimal, binary, or decimal format.

## 3.27 DATA DEFINITION INSTRUCTIONS

There are two data definitions instruction statements:
Define Constant (DC) and Define Storage (DS).

These statements are used to enter data constants into storage, and to define and reserve areas of storage. The statements may be named by symbols so that other program statements can refer to the fields generated from them. The discussion of the DC instruction is far more extensive than that of the DS instruction, because the DS instruction is written in the same format as the DC instruction and may specify some or all of the information that the DC instruction provides. Only the function and treatment of the statements vary. For this reason, the DC instruction is discussed first and in more detail than the DS instruction.

### 3.27.1 DC - Define Constant

The DC instruction is used to define data constants in storage. It may specify one constant or a series of constants, thereby relieving the programmer of the necessity to write a separate data definition statement for each constant desired. Furthermore, a variety of constants may be specified: fixed-point, floating-point, decimal, hexadecimal, character, and storage address. (Data constants are generally called constants unless they are created from storage addresses, in which case they are called address constants).

The typical form of the DC instruction statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| A symbol or blank | DC | One or more operands in the format described in the following text, each separated by commas. |

Each operand consists of four subfields; the first three describe the constant, and the fourth subfield provides the constant or constants. The first and third subfields may be omitted, the second and fourth must be specified, except for character type constants, in which case only field 4 is required. Note that more than one constant may be specified in the fourth subfield for most types of constants. Each constant so specified must be of the same type; the descriptive subfields that precede the constants apply to all of them. No blanks may occur within any of the subfields (unless provided as characters in a character constant or a character selfdefining term), nor may they occur between the subfields of an operand.

The subfields of the DC operand are written in the following sequence:

| Subfield | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| Duplication Factor | Type | Modifiers | Constant(s) |

Although the constants specified within one operand must have the same characteristics, the individual operand may be completely different (e.g., PL1'01',A(SYM)).

The symbol that names the DC instruction is the name of the constant (or first constant if the instruction specifies more than one). Relative addressing (e.g., SYMBOL+2) may be used to address the various constants if more than one has been specified, because the number of bytes allocated to each constant can be determined.

The value attribute of the symbol naming the DC instruction is the address of the leftmost byte (after any necessary alignment) of the first, or only, constant.

The length attribute depends on two things, the type of constant being defined and the presence of a length specification. Implied lengths are assumed for the various constant types in the absence of a length attribute. If more than one constant has been defined, the length attribute is the length in bytes (specified or implied) of the first constant.

Boundary alignment also varies according to the type of constant being specified and the presence of a length specification. Some constant types are only aligned to a byte boundary, but the DS instruction can be used to force any type of word boundary alignment for them. This is explained under "DS-Define Storage". Other constants are aligned at various word boundaries half, full, or double) in the absence of a length specification. If length is specified, no boundary alignment occurs for such constants.



Bytes that must be skipped in order to align the field at the proper boundary are not considered to be part of the constant. In other words, the Location Counter is incremented to reflect the proper boundary (if any incrementing is necessary) before the address value is established. Thus, the symbol naming the constant will not receive a value attribute that is the location of a skipped byte.

Any bytes skipped in aligning statements that do not cause information to be assembled are not zeroed. Thus, bytes skipped to align a DC statement are zeroed, and bytes skipped to align a DS statement are not zeroed.

Appendix A-1 summarizes, in chart form, the information concerning constants discussed in this section.

### 3.27.2 Operand Subfield 1: Duplication Factor

The duplication factor may be omitted. If specified, it causes the constant(s) to be generated the number of times indicated by the factor. The factor may be specified either by unsigned decimal self-defining term or by a positive absolute expression that is enclosed in parentheses. The duplication factor is evaluated after the constant is assembled. All symbols in the expression must be previously defined.

For DC constants the maximum value is 65535 ($=2^{16}-1$).

Note that a duplication factor of zero is permitted except in a literal and achieves the same result as it would in a DS instruction. See "Forcing Alignment" in section 3.29.2, "DS-Define Storage".

Note:

If duplication is specified for an address constant containing a Location Counter reference, the value of the Location Counter used in each duplication is incremented by the length of the operand.

### 3.27.3 Operand Subfield 2: Type

The type subfield defines the type of constant being specified. From the type specification, the assembler determines how it is to interpret the constant and translate it into the appropriate machine format. The type is specified by a single-letter code as shown in Figure 3-5. Note that this subfield may be omitted if the constant is a character type.

Further information about these constants is provided in the discussion of the constants themselves under "Operand Subfield 4: Constant".

### 3.27.4 Operand Subfield 3: Modifiers

Modifiers describe the length in bytes desired for a constant (in contrast to an implied length), and the scaling and exponent for the constant. If multiple modifiers are written, they must appear in this sequence: length, scale, exponent. Each is written and used as described in the following text.

**Length Modifier:** This is written as Ln, where n is either an unsigned decimal self-defining term or a positive absolute expression enclosed in parentheses. Any symbols in the expression must be previously defined.

The value of n represents the number of bytes of storage that are reserved for the constant. The maximum value permitted for the length modifiers supplied for the various types of constants is summarized in Appendix A-1. This table also indicates the implied length for each type of constant; the implied length is used unless a length modifier may be specified for any type of constant. However, no boundary alignment will be provided when a length modifier is given.

The Length must be greater than 0 (also for literals).

| Code | Type of Constant | Machine Format |
|---|---|---|
| C | Character | 8-bit code for each character |
| X | Hexadecimal | 4-bit code for each hexadecimal digit |
| B | Binary | Binary format |
| F | Fixed-point | Signed, fixed-point binary format; normally a full word |
| H | Fixed-point | Signed, fixed-point binary format; normally a half word |
| E | Floating-point | Short floating-point format; normally a full word |
| D | Floating-point | Long floating-point format; normally a double word |
| L | Floating-point | Extended floating point format; normally 2 double words |
| P | Decimal | Packed decimal format |
| Z | Decimal | Zoned decimal format |
| A | Address | Value of address; normally a full word |
| Y | Address | Value of address; normally a half word |
| S | Address | Base register and displacement value; a half word |
| V | Address | Space reserved for external symbol addresses; each address is normally a full word |
| Q | Address | Space reserved for dummy register offset from the beginning of the dummy register vector |

Table 3-5: Type Code For Constants

**Scale Modifier:** This modifier is written as Sn, where n is either a decimal value or an absolute expression enclosed in parentheses.

Any symbol in the expression must be previously defined. The decimal value or the parenthesized expression may be preceded by a sign; if none is present, a plus sign is assumed. The maximum values for scale modifiers are summarized in Appendix A-1.

A scale modifier may only be used with fixed-point (F,H) and floating-point (E,D,L) constants. It is used to specify the amount of internal scaling that is desired, as follows.

**Scale Modifier for Fixed-Point Constants:** The scale modifier specifies the power of two by which the constant must be multiplied after it has been converted to its binary representation. Just as multiplication of a decimal number by a power of 10 causes the decimal point to move, multiplication of a binary number by a power of two causes the binary point to move. This multiplication has the effect of moving the binary point away from its assumed position in the binary field; the assumed position being to the right of the rightmost position.

Thus, the scale modifier indicates either of the following:

1. the number of binary positions to be occupied by the fractional portion of the binary number, or

2. the number of binary positions to be deleted from the integral portion of the binary number.

A positive scale of x shifts the integral portion of the number x binary positions to the left, thereby reserving the rightmost x binary positions for the fractional portion. A negative scale shifts the integral portion of the number right, thereby deleting rightmost integral positions. If a scale modifier does not accompany a fixed-point constant containing a fractional part, the fractional part is lost.

In all cases where positions are lost because of scaling (or the lack of scaling), rounding occurs in the leftmost bit of the lost portion. The rounding is reflected in the rightmost position saved.

**Scale Modifier for Floating-point Constants:** Only a positive scale modifier may be used with a floating-point constant. It indicates the number of hexadecimal positions that the fraction is to be shifted to the right. Note that this shift amount is in terms of hexadecimal positions, each of which is four binary positions. (A positive scaling actually indicates that the point is to be moved to the left. However, a floating-point constant is always converted to a fraction, which is hexadecimally normalized. The point is assumed to be at the left of the leftmost position in the field. Since the point cannot be moved left, the fraction is shifted right).

Thus, scaling that is specified for a floating-point constant provides an assembled fraction that is unnormalized, i.e., contains hexadecimal zeros in the leftmost positions of the fraction. When the fraction is shifted, the exponent is adjusted accordingly to retain the correct magnitude. When hexadecimal positions are lost, rounding occurs in the left most hexadecimal position of the lost portion. The rounding is reflected in the rightmost hexadecimal position saved.

**Exponent Modifier:** This modifier is written in the form En, where n is either a decimal self-defining term or an absolute expression enclosed in parentheses. Any symbols in the expression must be previously defined. The decimal value or the parenthesized expression may be preceded by a sign; if none is present, a plus sign is assumed. The maximum values for exponent modifiers are summarized in Appendix A.1.

An exponent modifier may be used with fixed point (F,H) and floating-point (E, D, L) constants only. The modifier denotes the power of 10 by which the constant is to be multiplied before its conversion to the proper internal format.

This modifier is not to be confused with the exponent of the constant itself, which is specified as part of the constant and is explained under "Operand Subfield 4: Constant". Both are denoted in the same fashion, as En. The exponent modifier affects each constant in the operand, whereas the exponent written as part of the constant, only pertains to that constant. Thus, a constant may be specified with an exponent of +2, and an exponent modifier of +5 may precede the constant. In effect, the constant has an exponent of +7.

Note that there is a maximum value, both positive and negative, listed in Appendix A-1 for exponents. This applies to the exponent modifier and to the sum of the exponent modifier and the exponent specified as part of the constant.

### 3.27.5 Operand Subfield 4: Constant

This subfield supplies the constant (or constants) described by the subfields that precede it. A data constant (all types except A, Y, S, V and Q) is enclosed in apostrophes.
An address constant (types A, Y, S, V and Q) is enclosed in parentheses.
To specify two or more constants in the subfield, the constants must be separated by commas and the entire sequence of constants must be enclosed in the appropriate delimiters (i.e., apostrophes or parentheses). Thus, the format for specifying the constant(s) is one of the following:

| Single Constant | Multiple Constants (not permitted for character constants.) |
|---|---|
| 'constant' | 'constant, ...,constant' |
| (constant) | (constant, ...,constant) |

All constant types except character (C), hexadecimal (X), binary (B), packed decimal (P), and zoned decimal (Z), are aligned on the proper boundary, as shown in Appendix A-1, unless a length modifier is specified. In the presence of a length modifier, no boundary alignment is performed. If the operand specifies more than one constant, any necessary alignment applies to the first constant only. Thus, for an operand that provides five full-word constants, the first would be aligned on a full-word boundary, and the rest would automatically fall on full-word boundaries.

The total storage requirement of the operand is the product of the length times the number of constants in the operand times the duplication factor (if present) plus any bytes skipped for boundary alignment. If more than one operand is present, the storage requirement is derived by summing the requirements for each operand.

If an address constant contains a Location Counter reference, the Location Counter value that is used is the storage address of the first byte the constant will occupy. Thus, if several address constants in the same instruction refer to the Location Counter, the value of the Location Counter varies from constant to constant. Similarly if a single constant is specified (and it contains a Location Counter reference) with a duplication factor, the constant is duplicated with a varying Location Counter value.

E and H constants are converted as if they were D and F, respectively, and then shortened.

The subsequent text describes each of the constant types and provides examples.

**Character Constant-C:** Any of the valid 256 punch combinations may be designated in a character constant. Only one character constant may be specified per operand.

Special consideration must be given to representing apostrophes and ampersands as characters. Each apostrophe or ampersand desired as a character in the constant must be represented by a pair of apostrophes or ampersands. Only one apostrophe or ampersand appears in storage.

The maximum length modifier of a character constant is 256 bytes. No boundary alignment is performed. Each character is translated into one byte. Double apostrophes or double ampersands count as one character. If no length modifier is given, the size in bytes of the character constant is equal to the number of characters in the constant. If a length modifier is provided, the result varies as follows:

1. If the number of characters in the constant exceeds the specified length, as many rightmost bytes as necessary are dropped.

2. If the number of characters is less than the specified length, the excess rightmost bytes are filled with blanks.

In the following example, the length attribute of FIELD is 12:

| Name | Operation | Operand |
|---|---|---|
| FIELD | DC | C'TOTAL IS 110' |

However, in this next example, the length attribute is 15, and three blanks appear in storage to the right of the zero:

| Name | Operation | Operand |
|---|---|---|
| FIELD | DC | CL15'TOTAL IS 110' |

In the next example, the length attribute of FIELD is 12, although 13 characters appear in the operand. The two ampersands count as only one byte.

| Name | Operation | Operand |
|---|---|---|
| FIELD | DC | C'TOTAL IS &&10' |

Note that in the next example, a length of four has been specified, but there are five characters in the constant.

| Name | Operation | Operand |
|---|---|---|
| FIELD | DC | 3CL4'ABCDE' |

The generated constant would be:

ABCDABCDABCD

On the other hand, if the length had been specified as six instead of four, the generated constant would have been:

ABCDE␣ABCDE␣ABCDE␣

Note that the same constant could be specified as a literal.

| Name | Operation | Operand |
|---|---|---|
| | MVC | AREA(12),=3CL4'ABCDE' |

**Hexadecimal Constant-X:** A hexadecimal constant consists of one or more of the hexadecimal digits, which are 0-9 and A-F. The maximum length modifier of a hexadecimal constant is 256 bytes (512 hexadecimal digits).

Any word boundary alignment is only performed if a hexadecimal constant is used as a literal. The literal is aligned on half-word, full-word or double-word boundary if the explicit length is L2, L4, L8.

Constants that contain an even number of hexadecimal digits are translated as one byte per pair of digits. If an odd number of digits is specified, the leftmost byte has the leftmost four bits filled with a hexadecimal zero, while the rightmost four bits contain the odd (first) digit.

If no length modifier is given, the implied length of the constant is half the number of hexadecimal digits in the constant (assuming that a hexadecimal zero is added to an odd number of digits). If a length modifier is given, the constant is handled as follows:

1. If the number of hexadecimal digit pairs exceeds the specified length, the necessary leftmost bits (and/or bytes) are dropped.

2. If the number of hexadecimal digit pairs is less than the specified length, the necessary bits (and/or bytes) are added to the left and filled with hexadecimal zeros.

An eight-bit hexadecimal constant provides a convenient way of setting the bit pattern of a full binary word. The constant in the following example would set the first and third bytes of a word to 1's.

| Name | Operation | Operand |
|---|---|---|
| TEST | DS<br>DC | 0F<br>X'FF00FF00' |

The DS instruction sets the Location Counter to a full-word boundary.

The next example uses a hexadecimal constant as a literal and inserts 1's into low order of register 5.

| Name | Operation | Operand |
|---|---|---|
| | IC | 5,=X'FF' INSERT CHAR. |

In the following example, the digit A would be dropped, because five hexadecimal digits are specified for a length of two bytes:

| Name | Operation | Operand |
|---|---|---|
| ALPHA | DC | 3XL2'A6F4E' |

The resulting constant would be 6F4E, which would occupy the specified two bytes. It would then be repeated two more times, as requested by the duplication factor. If it had merely been specified as X'A6F4E', the resulting constant would have had a hexadecimal zero in the leftmost position:

0A6F4E.

In the following definition

| Name | Operation | Operand |
|---|---|---|
| TEST | DC | XL4'01020304' |

the constant will not be aligned. Nor will the literal =X'01020304'. The literal =XL4'01020304', however, will be aligned on word boundary.

**Binary Constant-B:** A binary constant is written using 1's and 0's enclosed in apostrophes. Duplication and length may be specified. The maximum length modifier of a binary constant is 256 bytes.

The implied length of a binary constant is the number of bytes occupied by the constant including any padding necessary. Padding or truncation takes place on the left. The padding bit used is a 0.

The following example shows the coding used to designate a binary constant. BCON would have a length attribute of one.

| Name | Operation | Operand |
|---|---|---|
| BCON | DC | B'11011101' |
| BTRUNC | DC | BL1'100100011' |
| BPAD | DC | BL1'101' |

BTRUNC would assemble with the leftmost bit truncated, as follows: 00100011

BPAD would assemble with five zeros as padding, as follows: 00000101

**Fixed-point Constants - F and H:**

A fixed-point constant is written as a decimal number, which may be followed by a decimal exponent if desired. The number may be an integer, a fraction, or a mixed number (i.e., one with integral and fractional portions). The format of the constant is as follows:

1. The number is written as a signed or unsigned decimal value. The decimal point may be placed before, within, or after the number, or it may be omitted, in which case the number is assumed to be an integer. A positive sign is assumed if an unsigned number is specified. Unless a scale modifier accompanies a mixed number of fraction, the fractional portion is lost, as explained under **SUBFIELD 3: Modifiers.**

2. The exponent is optional. If specified, it is written immediately after the number as En, where n is an optionally signed decimal value specifying the exponent of the factor 10. The exponent may be in the range -85 to +75. If an unsigned exponent is specified, a plus sign is assumed. The exponent causes the value of the constant to be adjusted by the specified power of 10 before the constant is converted to its binary form.

The exponent may be in the range -99 to +99, provided that the sum of the exponent and the exponent modifier do not exceed the permissible range for exponents.

The number is converted to a binary number. The binary number is then rounded and assembled into the proper field, according to the specified or implied length. If the value of the number exceeds the length specified or implied, the sign is lost, the necessary leftmost bits are truncated to the length of the field and the value is then assembled into the whole field. Any duplication factor that is present is applied after the constant is assembled. A negative number is carried in 2's complement form.

An implied length of four bytes is assumed for a full-word (F) and two bytes for a half-word (H), and the constant is aligned to the proper full-word or half-word boundary, if a length is not specified. However, any length up to and including eight bytes may be specified for either type of constant by a length modifier, in which case no boundary alignment occurs.

Maximum and minimum values, exclusive of scaling, for fixed-point constants are:

| Length | Max | Min |
|---|---|---|
| 8 | $2^{63}-1$ | $-2^{63}$ |
| 4 | $2^{31}-1$ | $-2^{31}$ |
| 2 | $2^{15}-1$ | $-2^{15}$ |
| 1 | $2^{7}-1$ | $-2^{7}$ |

A field of three full-words is generated from the statement shown below. The location attribute of CONWRD is the address of the left-most byte of the first word, and the length attribute is four, the implied length for a full-word fixed-point constant. The expression CONWRD+4 could be used to address the second constant (second word) in the field.

| Name | Operation | Operand |
|---|---|---|
| CONWRD | DC | 3F'658474' |

The next statement causes the generation of a two-byte field containing a negative constant. Notice that scaling has been specified in order to reserve six bits for the fractional portion of the constant.

| Name | Operation | Operand |
|---|---|---|
| HALFCON | DC | HS6'-25.46' |

The next constant (3.50) is multiplied by 10 to the -2 before being converted to its binary format. The scale modifier reserves twelve bits for the fractional portion.

| Name | Operation | Operand |
|---|---|---|
| FULLCON | DC | HS12'3.50E-2' |

The same constant could be specified as a literal:

| Name | Operation | Operand |
|---|---|---|
| | AH | 7,=HS12'3.50E-2' |

The final example specifies three constants. Notice that the scale modifier requests four bits for the fractional portion of each constant. The four bits are provided whether or not the fraction exists.

| Name | Operation | Operand |
|---|---|---|
| THREECON | DC | FS4'10,25.3,100' |

**Floating-point Constants (E,D,L):** A floating-point constant is written as a decimal number, which may be followed by a decimal exponent, if desired. The number may be an integer, a fraction, or a mixed number (i.e., one with integral and fractional portions). The format of the constant is as follows:

1. The number is written as a signed or unsigned decimal value. The decimal point may be placed before, within, or after the number, or it may be omitted, in which case the number is assumed to be an integer. A positive sign is assumed if an unsigned number is specified.

2. The exponent is optional. If specified, it is written immediately after the number as En, where n is an optionally signed decimal value specifying the exponent of the factor 10. The exponent may be in the range -99 to +99, provided that the sum of the exponent and the exponent modifier does not exceed the permissible range for exponents and the value of the expression is within the range of the representable numbers (see Appendix A.2 for the relevant estimates). If an unsigned exponent is specified, a plus sign is assumed.

Machine format for a floating-point number is in two parts: The portion containing the exponent, which is sometimes called the characteristic, followed by the portion containing the fraction, which is sometimes called the mantissa. Therefore, the number specified as a floating-point constant must be converted to a fraction before it can be translated into the proper format. For example, the constant 27.35E2 represents the number 27.35 * 10↑2↑. Represented as a fraction, it would be 0.2735 * 10↑4↑, the exponent having been modified to reflect the shifting of the decimal point. The exponent may also be affected by the presence of an exponent modifier, as explained under **Operand Subfield 3: Modifiers.**

The exponent is then translated into its binary equivalent, and the fraction is converted to a binary number. Scaling (see Appendix A.1: Range for Scale) is performed if specified; if not the fraction is normalized (leading hexadecimal zeros are removed). Rounding of the fraction is then performed according to the specified or implied length, and the number is assembled into the proper field. Within the portion of the floating-point field allocated to the fraction, the hexadecimal point is assumed to be the left of the leftmost hexadecimal digit, and the fraction occupies the leftmost portion of the field. Negative fractions are carried in true representation, not in the 2's complement form.

The lengths implied for full-word (E), double-word (D), and extended (L) constants are four, eight, and 16 bytes, respectively. The location counter is aligned at full-word (E) or double-word (D,L) boundary if a length is not specified. However, any length up to and including 8 bytes may be specified for constant types D,E by a length modifier; up to and including 16 bytes, for L-constants.

But no boundary alignment occurs in that case.

Any of the following statements could be used to specify 46.415 as a positive, full-word, floating-point constant; the last is a machine instruction statement with a literal operand. Note that the last two constants contain an exponent modifier.

| Name | Operation | Operand |
|---|---|---|
| | DC | E'46.415' |
| | DC | E'46415E-3' |
| | DC | E'+464.15E-1' |
| | DC | E'+.46415E+2' |
| | DC | EE2'.46415' |
| | AE | 6,=EE2'.46415' |

The following would each be generated as double-word floating-point constants.

| Name | Operation | Operand |
|---|---|---|
| FLOAT | DC | DE+4'+46,-3.729,+473' |

The following constants are each generated as floating-point constants in 2 double words:

| Name | Operation | Operand |
|---|---|---|
| EXTEND | DC | LE3'48,-2.71' |

**Decimal Constants-P and Z:** A decimal constant is written as a signed or unsigned decimal value. If the sign is omitted, a plus sign is assumed. The decimal point may be written wherever desired or may be omitted. Scaling and exponent modifiers may not be specified for decimal constants. The maximum length of decimal constant is 16 bytes. No word boundary alignment is performed. The position of a decimal point in the definition does not affect the assembly of the constant in any way, because, unlike fixed-point and floating-point constants, a decimal constant is not converted to its binary equivalent. The fact that a decimal constant is an integer, a fraction, or a mixed number is not pertinent to its generation. Furthermore, the decimal point is not assembled into the constant. The programmer may determine proper decimal point alignment either by defining his data so that the point is aligned or by selecting machine instructions that will operate on the data properly (i.e., shift it for purposes of alignment).

The Location Counter is aligned only if a decimal constant is used as a literal. The literal is aligned on half-word, full-word or double-word boundary if the length is L2, L4, L8.

If zoned decimal format is specified (Z), each decimal digit is translated into one byte. The translation is done according to the character set shown in Appendix A-3. The rightmost byte contains the sign as well as the rightmost digit. For packed decimal format (P), each pair of decimal digits is translated into one byte. The rightmost digit and the sign are translated into the rightmost byte. The bit configuration for the digits is identical to the configurations for the hexadecimal digits 0-9 as explained under "Hexadecimal Self-Defining Value". For both packed and zoned decimals, a plus sign is translated into the hexadecimal digit C, and a minus sign into the digit D.

If an even number of packed decimal digits is specified, one digit will be left unpaired, because the rightmost digit is paired with the sign. Therefore, in the leftmost byte, the leftmost four bits will be set to zeros and the rightmost four bits will contain the odd (first) digit.

If no length modifier is given, the implied length for either constants is the number of bytes the constant occupies (taking into account the format, sign, and possible addition of zero bits for packed decimals). If a length modifier is given, the constant is handled as follows:

1. If the constant requires fewer bytes than the length specifies, the necessary number of bytes is added to the left. For zoned decimal format, the decimal digit zero is placed in each added byte. For packed decimals, the bits of each added byte are set to zero.

2. If the constant requires more bytes than the length specifies, the necessary number of leftmost digits or pairs of digits is dropped, depending on which format is specified.

Examples of decimal constants definition:

| Name | Operation | Operand |
|---|---|---|
| | DC | P'+1.25' |
| | DC | Z'-543' |
| | DC | Z'79.68' |
| | DC | PL3'79.68' |

The following statement specifies three packed decimal constants. The length modifier applies to each packed decimal constant.

| Name | Operation | Operand |
|---|---|---|
| DECIMALS | DC | PL8'+25.8,-3874,+2.3' |



The last example illustrates the use of a packed decimal literal.

| Name | Operation | Operand |
|---|---|---|
| | UNPK | OUTAREA,=PL2'+25' |

### 3.27.6 Address Constants

An address constant is a virtual storage address or an absolute expression that is translated into a constant. Address constants are normally used for initializing base registers to facilitate the addressing of storage. Furthermore, they provide the means of communicating between control sections of a multi-section program. However, storage addressing and control section communication are also dependent on the use of the USING assembly instruction and the loading of registers. Coding examples that illustrate these considerations are contained in the discussion of "Programming with the USING instruction" (section 3.12.2).

An address constant, unlike other types of constants, is enclosed in parentheses. If two or more address constants are specified in a statement, they are separated by commas, and the entire sequence enclosed in parentheses. There are five types of address constants: A, Y, S, V and Q.

**Complex Relocatable Expressions:** A complex relocatable expression can only be used in an A-type or Y-type address constant. These expressions contain two or more **unpaired** relocatable terms and/or a negative relocatable term in addition to any absolute or paired relocatable terms that may be present. A complex relocatable expression might consist of external symbols (which cannot be paired) and designate an address in an independent assembly that is to be linked and loaded with the assembly containing the complex relocatable expression.

The value of the expression is determined when the referenced control sections are loaded. Complex relocatable expressions can be used to determine the distance between two control sections after they are loaded into main storage.

**A-type Address Constant:** This constant is specified as an absolute, relocatable, or complex relocatable expression. (Remember that an expression may be single term or multiterm.) The value of the expression is calculated to 32 bits with one exception: the maximum value of the expression may be $2^{32}-1$. The value is then truncated on the left, if necessary, to the specified or implied length of the field and assembled into the rightmost bits of the field. The implied length of an A-type constant is four bytes and alignment is to a full-word boundary unless a length is specified, in which case no alignment will occur. A length of 1-4 bytes may be specified. The programmer must remember that the value of relocatable A-constants may be truncated if fewer than 4 bytes are used as the length modifier.

In the following examples, the field generated from the statement named ACONST contains four constants, each of which occupies four bytes. Note that there is a Location Counter reference in one.

The value of the Location Counter will be the address of the first byte allocated to the fourth constant. The second statement shows the same set of constants specified as literals (i.e., address constant literals).

| Name | Operation | Operand |
|---|---|---|
| ACONST | DC<br>LM | A(108,LOOP,END-STRT,*+4096)<br>4,7,=A(108,LOOP,END-STRT,*+4096) |

Note:

When the Location Counter reference occurs in a literal, as in the LM instruction above, the value of the Location Counter is the address of the first byte of the instruction.

**Y-type Address Constant:** A Y-type address constant has much in common with the A-type constant. It, too, is specified as an absolute, relocatable, or complex relocatable expression. The value of the expression is also calculated to 32 bits. However, the maximum value of the expression may be only $2^{16}-1$. The value is then truncated, if necessary, to the specified or implied length of the field and assembled into the rightmost bits of the field. The implied length of a Y-type constant is two bytes and alignment is to a half-word boundary unless a length is specified, in which case no alignment occurs. The maximum length of a Y-type address constant is two bytes.

**S-type Address Constant:** The S-type address constant is used to store an address in base-displacement form.

The constant may be specified in two ways:

1. As an absolute or relocatable expression, e.g., S(BETA)

2. As two absolute expressions, the first of which represents the displacement value and the second, the base register, e.g., S(400(13)).

The address value represented by the expression in (1) will be broken down by the assembler into the proper base register and displacement value. An S-type constant is assembled as a halfword and aligned on a halfword boundary. The leftmost four bits of the assembled constant represent the base register designation, the remaining 12 bits the displacement value.

If length specification is used, only two bytes may be specified. S-type address constants may not be specified as literals.

**V-type Address Constant:** This constant is used to reserve storage for the address of an external symbol that is used for effecting branches to other programs. The constant may not be used for external data references. The constant is specified as one relocatable symbol, which must not be identified by an EXTRN statement. The symbol used is assumed to be an external symbol by virtue of the fact that it is supplied in a V-type constant.

Note that specifying a symbol as the operand of a V-type constant does not constitute a definition of the symbol for this assembly. The implied length of a V-type address constant is four bytes, and boundary alignment is to a full word. A length modifier may be used to specify a length of either three or four bytes, in which case no such boundary alignment occurs, and truncation of the relocated value may result. In the following example, 12 bytes will be reserved, because there are three symbols. The value of each assembled constant will be zero until the program is loaded.

| Name | Operation | Operand |
|---|---|---|
| VCONST | DC | V(SORT,MERGE,CALC) |

**Q-type Address Constant:** The Q-type Address Constant is used to reserve storage space for the offset of a dummy register, which must previously have been defined as a DXD or DSECT. If it has been defined as a DSECT, this DSECT is taken as a dummy register in relation to the external information. The implied length of a Q-type constant is four bytes. Length specifications of 1 to 4 are legal.

In the following example, D1 and D2 have previously been defined as DXD or DSECT. 8 bytes are reserved for the Q-type constants by means of two symbols.

The value of the constants, i.e. the offset of the dummy registers from the beginning of the dummy register vector, is entered by the Linkage Editor.

| Name | Operation | Operand |
|---|---|---|
| QCONST | DC | Q(D1,D2) |

Example:

```
DC Q(DXDEXT)
```

## 3.28 LITERAL DEFINITIONS

The reader is reminded that the discussion of literals as machine instruction operands referred him to the description of the DC operand for the method of writing a literal operand. All subsequent operand specifications are applicable to writing literals, the only differences being:

1. The literal is preceded by an = sign.

2. The duplication factor may not be zero.

3. S-type address constants may not be specified.

4. Multiple operands may not be specified.

Examples of literals appear throughout the balance of the DC instruction discussion.

# 3.29 DS INSTRUCTION

## 3.29.1 DS - Define Storage



The DS instruction is used to reserve areas of storage and to assign names to those areas. The use of this instruction is the preferred way of symbolically defining storage for work areas, input/output areas, etc.

The typical form of the DS statement is:

| Name | Operation | Operand |
|---|---|---|
| A symbol or blank | DS | One or more operands in the format described in the following text, each separated by commas. |

The format of the DS operand is identical to that of the DC operand, exactly the same subfields are employed, written in exactly the same sequence as they are in the DC operand. Although the formats are identical, there are two differences in the specification of subfields. They are:

1. The specification of data (subfield 4) is optional in a DS operand, but it is mandatory in a DC operand. If a constant is specified, it must be valid.

2. The maximum length that may be specified for character (C) and hexadecimal (X) field types is 65,535 bytes rather than 256 bytes.

3. The maximum duplication factor for DS constants is 16777215 (= $2^{24}-1$).

If a DS operand specifies a constant in subfield 4, and no length is specified in subfield 3, the assembler determines the length of the data and reserves the appropriate amount of storage. **It does not assemble the constant.** It is convenient for the programmer to be able to specify data and have the assembler calculate the storage area that would be required for such data. If he knows the general format of the data that will be placed in the storage area during program execution, all he needs to do is specify it in the fourth subfield in a DS operand. The assembler then determines the correct amount of storage to be reserved, thus relieving the programmer of the need to specify length.

If the DS instruction is named by a symbol, its value attribute is the location of the leftmost byte of the reserved area. The length attribute of the symbol is determined in the same manner as for a DC. Any positioning required for aligning the storage area to the proper type of boundary is done before the address value is determined. Bytes skipped for alignment are not set to zero.

Each field type (e.g., hexadecimal, character, floating-point) is associated with certain characteristics (these are summarized in Appendix A.1). The associated characteristics will determine which field-type code the programmer selects for the DS operand and what other information he adds, notably a length specification or a duplication factor. For example, the E floating-point field and the F fixed-point field both have an implied length of four bytes. The leftmost byte is aligned to a full-word boundary. Thus, either code could be specified if it were desired to reserve four bytes of storage aligned to a full-word boundary. To obtain a length of eight bytes, one could specify either the E or F field type with a length modifier of eight. However, a duplication factor would have to be used to reserve a larger area, because the maximum length specification for either type is eight bytes. Note also that specifying length would cancel any special boundary alignment.

In contrast, packed and zoned decimal (P and Z), character (C), hexadecimal (X), and binary (B) fields have an implied length of one byte. Any of these codes, if used, would have to be accompanied by a length modifier, unless just one byte is to be reserved. Although no alignment occurs, the use of C and X field types permits greater latitude in length specifications.

Unless a field of one byte is desired, either the length must be specified for the C, X, P, Z, or B field types, or else the data must be specified (as the fourth subfield), so that the assembler can calculate the length.

To define four 10-byte fields and one 100-byte field, the respective DS statements might be as follows:

| Name | Operation | Operand |
|---|---|---|
| FIELD | DS | 4CL10 |
| AREA | DS | CL100 |

Although FIELD might have been specified as one 40-byte field, the preceding definition has the advantage of providing FIELD with a length attribute of 10. This would be pertinent when using FIELD as an SS machine instruction operand.

Additional examples of DS statements are shown below:

| Name | Operation | Operand | |
|---|---|---|---|
| ONE | DS | CL80 | (one 80-byte field, length attribute of 80) |
| TWO | DS | 80C | (80 one-byte fields, length attribute of 1) |
| THREE | DS | 6F | (Six full words, length attribute of four) |
| FOUR | DS | D | (one double word, length attribute of eight) |
| FIVE | DS | 4H | (four half-words, length attribute of two) |

Note:

A DS statement causes the storage area to be reserved but not set to zeros. No assumption should be made as to the contents of the reserved area.

### 3.29.2 Special Uses of the Duplication Factor

**Forcing Alignment:** The Location Counter can be forced to a double-word, full-word, or half-word boundary by using the appropriate field type (e.g., D, F, or H) with a duplication factor of zero. This method may be used to obtain boundary alignment that otherwise would not be provided. For example, the following statements would set the Location Counter to the next double-word boundary and then reserve storage space for a 128-byte field (whose leftmost byte would be on a double-word boundary).

| Name | Operation | Operand |
|---|---|---|
| | DS | 0D |
| AREA | DS | CL128 |

**Defining Fields of an Area:** A DS instruction with a duplication factor of zero can be used to assign a name to an area of storage without actually reserving the area. Additional DS and/or DC instructions may then be used to reserve the area and assign names to fields within the area (and generate constants if DC is used).

For example, assume that 80-character records are to be read into an area for processing and that each record has the following format:

Positions 5 - 10 Payroll Number
Positions 11 - 30 Employee Name
Positions 31 - 36 Date
Positions 47 - 54 Gross Wages
Positions 55 - 62 Withholding Tax

The following example illustrates how DS instructions might be used to assign a name to the record area, then define the fields of the area and allocate the storage for them. Note that the first statement names the entire area by defining the symbol RDAREA: the statement gives RDAREA a length attribute of 80 bytes, but does not reserve any storage. Similarly, the fifth statement names a 6-byte area by defining the symbol DATE: the three subsequent statements actually define the fields of DATE and allocate storage for them. The second, ninth, and last statements are used for spacing purposes and, therefore, are not named.

| Name | Operation | Operand |
|---|---|---|
| RDAREA | DS | 0CL80 |
| | DS | CL4 |
| PAYNO | DS | CL6 |
| NAME | DS | CL20 |
| DATE | DS | 0CL6 |
| DAY | DS | CL2 |
| MONTH | DS | CL2 |
| YEAR | DS | CL2 |
| | DS | CL10 |
| GROSS | DS | CL8 |
| FEDTAX | DS | CL8 |
| | DS | CL18 |

## 3.30 DUMMY REGISTERS

Dummy registers are memory areas supporting the communication between various control sections. They can be defined anywhere in the program, i.e., the dummy register definitions need not be written consecutively. It is also possible to define one and the same dummy register, with differing lengths, in more than one assembly unit (see PSREG2 in Example 1).

The assembler calculates alignment and length of the dummy register and passes this information, via ESD entries, on to the Linkage Editor. During linkage editing, all dummy registers from all modules to be linked are combined to form one so-called "dummy register vector", i.e., the dummy registers all appear in the linked program in a specific consecutive order, defined by the Linkage Editor. Hence the suffix "vector". During linkage editing, the strongest attributes for identical dummy registers are used as the ultimately valid attributes for this dummy register. In Example 1 (see Appendix A.11), PSREG2 in CSECT 1 has a length of 60 bytes; in CSECT2, a length of 22 bytes. So after linkage editing, the dummy register PSREG2 has a length of 60 bytes. Exactly the same conditions prevail as far as alignment is concerned: if a dummy register had been aligned on halfword boundary in one CSECT, but on fullword boundary in another, this dummy register would have been aligned on fullword boundary after linking.

The maximum length of a dummy register is 4095 bytes.

The use of dummy registers has the following advantage over the application of common areas:

```
Assume   CSECT1 with PSREG1 and PSREG2
         CSECT2 with PSREG3 and PSREG4
```

Both CSECTS are to be linked into one program.

Within a CSECT it is only possible to access dummy registers which have been defined in this very CSECT. So in CSECT1, only PSREG1 and PSREG2. Intentional or inadvertent overwriting of PSREG3 and PSREG4 is impossible.

If the number of dummy registers is changed within a module, this module only will have to be reassembled and the entire program relinked. However, if you change the DSECT of a common area, then all modules affected by a change of contents or length of the common area have to be reassembled.

Note

Currently, the DLL does not support the use of dummy registers, i.e., programs must be linked with TSOSLNK and no dynamic loading mechanism can be used.

Dummy registers can be addressed in two ways:

- Via Q-type constants

  At program run-time the Q-type constant indicates the dummy register offset from the beginning of the dummy register vector. This means that this value must be added to the start address of the dummy register vector in order to address the dummy register.

- Via a base register

  The instruction

  USING *PRV,R8

  causes all instructions concerning dummy registers to use register 8 as the base register, irrespective of any other valid base registers. Register 8 must then be loaded with the start address of the dummy register vector.

### 3.30.1 Define Dummy Registers (DSECT, DXD)

A dummy register can be defined in two ways:

- As a DSECT, the name of the DSECT then being used as the operand of a Q-type constant.

| Name | Operation | Operand |
|---|---|---|
| NAMED | DSECT | Not used |
| . | | |
| . | | |
| QNAMED | DC | Q(NAMED) |

- Via the assembler instruction DXD

Format of the DXD instruction

| Name | Operation | Operand |
|---|---|---|
| NAME | DXD | Same operand as described for DS |

For examples see Appendix A.11

### 3.30.2 Define Length of Dummy Register Vector (CXD)

The CXD instruction reserves a word (four bytes) in which the Linkage Editor enters the total length of the dummy register vector.
The length of the dummy register vector is important for later processing, e.g., for allocation of the necessary memory space.

Format of the CXD instruction

| Name | Operation | Operand |
|---|---|---|
| NAME or blank | CXD | Blank |

The CXD instruction must be aligned on fullword boundaries and may occur anywhere in the program. The length attribute is four. The type attribute is A (see 4.15.1).

For an example see Appendix A.11 (Example 2).

## 3.31 LISTING CONTROL INSTRUCTIONS

The listing control instructions are used to identify an assembly listing and assembly output cards, to provide blank lines in an assembly listing, and to designate how much detail is to be included in an assembly listing. No instructions or constants are generated for the object program. Listing control statements except PRINT are never printed. The line count per page can be controlled using the LINECNT parameter.



### 3.31.1 TITLE - Identify Assembly Output

The TITLE assembly instruction enables the programmer to identify assembly listing and assembly output cards.

The TITLE instruction statement has the following format:

| Name | Operation | Operand |
|---|---|---|
| Name, sequence symbol, or blank | TITLE | One to 97 characters, enclosed in single apostrophes |

The name entry may contain a name of from one to four alphabetic or numeric characters in any combination. The contents of the name entry are output in columns 73-76 of the object module except those produced by the PUNCH and REPRO assembly instructions. Only the first TITLE statement in a program may have a name in the name entry. The name field of all subsequent TITLE statements must be blank.

The operand field may contain up to 97 characters enclosed in apostrophes. Any ampersands or apostrophes enclosed within the surrounding apostrophes must be represented by two ampersands or apostrophes. However, only one ampersand or apostrophe is printed and counted in the total number of operand character. The contents of the operand field are printed at the top of each page of the assembly listing.

A program may contain more than one TITLE statement. Each TITLE statement provides the heading for pages in the heading for pages in the assembly listing that follow it, until another TITLE statement is encountered. Each TITLE statement encountered after the first one causes the listing to be advanced to a new page (before the heading is printed).

For example, if the following statement is the first TITLE statement to appear in a program:

| Name | Operation | Operand |
|---|---|---|
| PGM1 | TITLE | 'FIRST HEADING' |

then PGM1 is punched into all of the output cards (columns 73-76) and this heading appears at the top of each page: FIRST HEADING.

If the following statement occurs later in the same program:

| Name | Operation | Operand |
|---|---|---|
| | TITLE | 'A NEW HEADING' |

then, PGM1 is still output in the output cards, but each following page begins with the heading: A NEW HEADING.

The sequence number of the cards in the output deck is contained in columns 77-80, except those produced by the PUNCH and REPRO assembly instructions.

Note:

If a TITLE statement is generated by means of a macro, it is also effective if the listing is deactivated by means of PRINT NOGEN. In this case it can be rendered ineffective by means of *COMOPT NOPRTIT (page feed, print header line).

### 3.31.2 EJECT - Start New Page

The EJECT assembly instruction causes the next line of the listing to appear at the top of a new page. This instruction provides a convenient way of separating routines in the program listing.

The EJECT Instruction statement has the following format:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | EJECT | Not used; should be blank. |

If the next line of the listing would appear at the top of a new page without the EJECT instruction, the EJECT instruction has no immediate effect. Two EJECTSs in succession cause a page to be skipped. A TITLE statement followed immediately by an EJECT statement will result in a page with a title line and a statement heading line. Text following the EJECT instruction will begin at the top of the next page.

### 3.31.3 SPACE - Space Listing

The SPACE assembly instruction is used to insert one or more blank lines in the listing.

The SPACE instruction statement has the following format:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | SPACE | A decimal value or blank |

A decimal value is used to specify the number of blank lines to be inserted in the assembly listing. A blank operand causes one blank line to be inserted. If this value exceeds the number of lines remaining on the listing page, the statement will have the same effect as an EJECT statement.

### 3.31.4 PRINT - Print Optional Data

The PRINT assembly instruction controls the content of the assembly listing.

The PRINT instruction has the following format:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | PRINT | One to twelve operands |

One to twelve operands of the following form may be used:

| | |
|---|---|
| BASE | After every USING and DROP statement, the particular addressable area for registers employed as base registers with USING is printed. |
| or<br>NOBASE | No areas for the base registers are printed. |
| CLOSED | - The cross-reference listing is single-spaced. |
| or<br>OPEN | - The cross-reference listing is double-spaced. |
| CODE | - The effect of PRINT NOGEN on statements generated by macro instructions is restricted. Only the code generated is output, the source line being suppressed. |
| or<br>NOCODE | - The full effect of PRINT NOGEN is retained. |
| COPY | Copied statements are listed. |
| or<br>NOCOPY | Copied statements are not listed. |

| | |
|---|---|
| DATA<br>or<br>NODATA | - Constants are printed out in full in the listing.<br>- Only the leftmost eight bytes (16 hexadecimal digits) are printed. |
| DECK<br>or<br>NODECK | - The object code is generated.<br>- Generation of the object code is interrupted. |
| GEN<br>or<br>NOGEN | - All statements generated by macro instructions are printed.<br>- Statements generated by macro instructions are not printed, except MNOTE messages which print regardless of NOGEN. However, the outer macro instruction itself will appear in the listing. |
| NDBF<br>or<br>DBF | - The normal assembly listing headings are generated.<br>- Headings for TRANSDATA 960 (DUET) instructions are generated. The DUET instructions are represented by bits. |
| NUM<br>or<br>NONUM | - Print the card number of the various object program card types.<br>- Inhibit printing the card number of the various card types. |
| ON<br>or<br>OFF | - A listing is printed.<br>- No listing is printed. |
| xON<br>or<br>xOFF | - Symbols beginning with the character x are included in the cross-reference listing. Up to 5 different entries are allowed.<br>- Symbols beginning with the character x are not included in the cross-reference listing. |
| REF<br>or<br>NOREF | - All symbols are included in the cross-reference listing.<br>- Only the referenced symbols are included in the cross-reference listing.<br>Note: The PRINT Statement operands for the cross-reference listing only cause the last statements to be executed. |
| SINGLE<br>or<br>DOUBLE | - Text listing is single spaced.<br>- Text listing is double spaced. |

A program may contain any number of PRINT statements. The conditions set by a PRINT statement are in effect until another PRINT statement is encountered.

If an operand is omitted, it is assumed to be unchanged and continuous according to its last specification.

When OFF is specified, GEN and DATA have no effect. When NOGEN is specified, DATA has no effect for generated constants.

Until the first PRINT statement is encountered, the following is assumed:

| Name | Operation | Operand |
|---|---|---|
| | PRINT | ON,NODATA,GEN |

For example, if the statement:

| Name | Operation | Operand |
|---|---|---|
| | DC | XL256'00' |

appears in a program, 256 bytes of zeros are assembled.

If the statement:

| Name | Operation | Operand |
|---|---|---|
| | PRINT | DATA |

is the last PRINT statement to appear before the DC statement, all 256 bytes of zeros are printed in the assembly listing. However, if there are no previous PRINT statements, or:

| Name | Operation | Operand |
|---|---|---|
| | PRINT | NODATA |

is the last PRINT statement to appear before the DC statement, only eight bytes of zeros are printed in the assembly listing.

## 3.32 PROGRAM CONTROL INSTRUCTIONS

The program control instructions are used:
- to specify the end of an assembly,
- to set the Location Counter to a value or half-word boundary,
- to insert previously written coding in the program,
- to specify the position of literals in storage,
- to check the sequence of input cards,
- to indicate statement format,
- to output cards to the object module.

Except for the CNOP instruction, none of these assembly instructions generate instructions or constants in the object program.

### 3.32.1 ICTL - Input Format Control

The ICTL instruction allows the programmer to alter the normal format of his source program statements. The ICTL statement must precede all other statements in the source program and may be used only once.

The ICTL instruction statement has the following format:

| Name | Operation | Operand |
|---|---|---|
| not used | ICTL | 1-3 decimal values of the form b,e,c |

Operand b specifies the begin column of the source statement. It must always be specified, and must be from 1-40, inclusive. Operand e specifies the end column of the source statement. The end column, when specified, must be from 41-79, inclusive; when not specified, it is assumed to be 71. The column after the end column is used to indicate whether the next card is a continuation card. Operand c specifies the continue column of the source statement. The continue column, when specified, must be from 2-40 and must be greater than or equal to the value of b. If the continue column is not specified, the assembler assumes that there are no continuation cards.

The next example designates the begin column as column 25. Since the end column is not specified, it is assumed to be column 71. No continuation cards are recognized because the continue column is not specified.

| Name | Operation | Operand |
|---|---|---|
| | ICTL | 25 |

### 3.32.2 ISEQ - Input Sequence Checking

The ISEQ instruction is used to check the sequence of input cards.

The ISEQ instruction has the following format:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | ISEQ | Two decimal values of the form l,r, or not used |

The operands l and r, respectively, specify the leftmost and rightmost columns of the field in the input cards to be checked. Operand r must be equal to or greater than operand l. Columns to be checked must not be between the "begin" and "end" column.

The field to be checked must not exceed 8 bytes. Sequence checking begins with the first card following the ISEQ statement. Comparison of adjacent cards makes use of the eight-bit internal collating sequence. Each card checked must be higher than the preceding one.

An ISEQ statement with a blank operand terminates the operation. Checking may be resumed with another ISEQ statement.

Sequence checking is only performed on statements contained in the source program. Statements generated by a macro instruction are not checked for sequence.

### 3.32.3 PUNCH - Output in Card Format

The PUNCH assembly instruction causes the data in the operand to be output in card format to the object module. Each PUNCH statement produces a card. As many PUNCH statements may be used as are necessary.

The PUNCH instruction has the following format:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | PUNCH | 1 to 80 characters enclosed in apostrophes |

The operand is a self-defining term of up to 80 characters. All characters, including blank, are valid. The position immediately to the right of the left apostrophe is mapped in column one of the card. The assembly program does not process the data other than causing it to be stored in card format. For each apostrophe to be mapped on the card and for each &-character, two apostrophes or &-characters must be specified in the operand. The two apostrophes or &-characters are reduced to a single apostrophe or &-character and count as only one character in the operand.

PUNCH statements may occur anywhere within a program or macro definition. If a PUNCH statement occurs before the first control section, the resultant card will precede all other cards in the object module; otherwise the card will be punched in place. No sequence number or identification is output to the card.

Note:

Unlike the BS2000 command PUNCH (see 2.7.4) this statement does not cause cards to be punched.

### 3.32.4 REPRO - Reproduce Following Card

The REPRO assembly instruction causes data on the following statement line to be output to the object module. The data is not processed; no substitution is performed for variable symbols. No sequence number or identification is output to the card. One REPRO instruction produces one card.

REPRO statements that occur before all statements composing the first or only control section will produce cards which precede all cards of the object program in the object module.

The REPRO instruction has the following format:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | REPRO | Not used, should not be present |

The line to be reproduced may contain any combination of up to 80 characters. Characters may be entered starting in column 1 and continue through column 80 of the line. Column 1 of the line is mapped in column 1 of the card.

### 3.32.5 ORG - Set Location Counter

The ORG instruction is used to alter the setting of the Location Counter for the current control section.

The ORG instruction has the following format:

| Name | Operation | Operand |
|---|---|---|
| A symbolic address, a sequence symbol or blank | ORG | An expression or blank |

Any symbols in the expression must have been previously defined. Any unpaired relocatable symbols must be defined in the same control section in which the ORG statement appears. The expression must be relocatable and must not be complex. Absolute expressions are invalid.

The Location Counter is set to the value of the expression in the operands. If the operand is omitted, the Location Counter is set to a location that is one byte higher than the maximum location assigned for the control section up to this point.



An ORG statement must not be used to specify a location below the beginning of the control section in which it appears. For example, the statement:

| Name | Operation | Operand |
|---|---|---|
| | ORG | *-500 |

is invalid if it appears less than 500 bytes from the beginning of the current control section.

If the Location Counter is to be reset to the next available location in the current control section, the following statement would be used:

| Name | Operation | Operand |
|---|---|---|
| | ORG | |

If previous ORG statements have reduced the Location Counter for the purpose of redefining a portion of the current control section, an ORG statement with an omitted operand can then be used to terminate the effects of such statements and restore the Location Counter to its highest setting.

A symbol in the name field of an ORG statement receives the address before execution of this statement. This provides the possibility to refer symbolically to the Location Counter before execution of the ORG statement.

Example:

| Name | Operation | Operand |
|---|---|---|
| | . | |
| | . | |
| | . | |
| NEWADDR | DS | CL100 |
| | . | |
| | . | |
| | . | |
| OLDADDR | ORG | NEWADDR |
| | . | |
| | . | |
| | . | |
| | ORG | OLDADDR |
| | . | |
| | . | |
| | . | |

### 3.32.6 LTORG - Begin Literal Pool

The LTORG assembly instruction causes all literals since the previous LTORG or beginning of the program to be assembled at appropriate boundaries starting at the first double-word boundary following the LTORG statement. If no literals follow the LTORG statement, alignment of the next instruction will occur. Bytes skipped are not zeroed. The maximum number of permitted LTORG statements is 255.

The literals are aligned as strongly as possible corresponding to their type and length. For example, the literals CL4'1234' and =C'1234' are both filed at word boundaries. An exception to this are the constants B, X, P, Z with implicit lengths, since in these cases a literal can contain several constants of different length.

The LTORG instruction has the following format:

| Name | Operation | Operand |
|---|---|---|
| A symbol, sequence symbol, or blank | LTORG | Not used, should not be present |

The symbols represents the address of the first byte of the literal pool. It has a length attribute of one.

#### Special Addressing Considerations

Any literals used after the last LTORG statement in a program are placed at the end of the first control section. If there are no LTORG statements in a program, all literals used in the program are placed at the end of the first control section. In these circumstances the programmer must ensure that the first control section is always addressable. This means that the base address register for the first control section should not be changed through usage in subsequent control sections. If the programmer does not wish to reserve a register for this purpose, he may place a LTORG statement at the end of each control section, thereby ensuring that all literals appearing in that section are addressable.

If duplicate literals occur within the range controlled by one LTORG statement, only one literal is stored. Literals are considered duplicates only if their specifications are identical. A unique literal will be generated even if it appears to duplicate another literal, if it is an A-type address constant containing a reference to the location counter.

The following examples illustrate how literals are stored if each pair occurs under one LTORG statement:

```
A(*+4)      Both are stored
A(*-4)
X'FFFF'     The first is stored
X'FFFF'
XL3'0'      Both are stored
HL3'0'
```

### 3.32.7 CNOP - Conditional No Operation

The CNOP instruction allows the programmer to align an instruction at a specific word boundary. If any bytes must be skipped in order to align the instruction properly, the assembler insures an unbroken instruction flow by generating no-operation instructions. This facility is useful when calling sequences to establish a linkage to a subroutine and when parameter sequences are to be created.

The CNOP instruction insures the alignment of the Location Counter setting to a half-word, word, or double-word boundary. If the Location Counter is already properly aligned, the CNOP instruction has no effect. If the specified alignment requires the Location Counter to be incremented, one to three no-operation instructions are generated, each of which uses two bytes.

The CNOP instruction statement has the following format:

| Name | Operation | Operand |
|---|---|---|
| Symbol, sequence symbol or blank | CNOP | Two absolute expressions of the form b,w |

Operand b specifies at which byte in a word or double word the Location Counter is to be set; b can be 0, 2, 4, or 6. Operand w specifies whether byte b is in a word (w=4) or double word (w=8).
The following pairs of b and w are valid:

| b,w | Specifies |
|---|---|
| 0,4 | Beginning of a word |
| 2,4 | Middle of a word (second half word) |
| 0,8 | Beginning of a double word |
| 2,8 | Second half word of a double word |
| 4,8 | Middle (third half word or second word) of a double word |
| 6,8 | Fourth half word of a double word |

| Double word | | | | | | | |
|---|---|---|---|---|---|---|---|
| Word | | | | Word | | | |
| Half word | | Half word | | Half word | | Half word | |
| Byte | Byte | Byte | Byte | Byte | Byte | Byte | Byte |
| 0,4<br>0,8 | | 2,4<br>2,8 | | 0,4<br>4,8 | | 2,4<br>6,8 | |

Table 3-7: CNOP Alignment of the location counter

Table 3-7 shows the position in a double word that each of these pairs specifies. Note that both 0,4 and 2,4 specify two different locations in a double word.

Assuming that the Location Counter is currently aligned at a double word boundary, then the CNOP instruction in this sequence:

| Name | Operation | Operand |
|---|---|---|
| | CNOP<br>BALR | 0,8<br>2,14 |

has no effect. However, this sequence:

| Name | Operation | Operand |
|---|---|---|
| | CNOP<br>BALR | 6,8<br>2,14 |

causes three branch-on-conditions (no-operations) to be generated, thus aligning the BALR instruction at the last half word in a double word as follows:

| Name | Operation | Operand |
|---|---|---|
| | BCR<br>BCR<br>BCR<br>BALR | 0,0<br>0,0<br>0,0<br>2,14 |

After the BALR instruction is generated, the Location Counter is at a double word boundary, thereby insuring an unbroken instruction flow.

Note:

If the Location Counter is on an odd-numbered byte-boundary when a CNOP instruction is encountered, normal alignment occurs before the CNOP is processed.

### 3.32.8 END - End Assembly

The END instruction terminates the assembly of a program. It may also designate a point in the program or in a separately assembled program to which control may be transferred after the program is loaded.

Points in separately assembled programs must be additionally identified in an EXTRN statement or as a V-type constant. The END instruction must always be the last statement in the source program.

The format of the END instruction statement is as follows:

| Name | Operation | Operand |
|---|---|---|
| A sequence symbol or blank | END | A relocatable expression or blank |

The operand specifies the point to which control may be transferred when loading is complete.

For example:

| Name | Operation | Operand |
|---|---|---|
| NAME | CSECT | |
| AREA | DS | 50F |
| BEGIN | BALR | 2,0 |
| | USING | *,2 |
| | . | |
| | . | |
| | . | |
| | END | BEGIN |

### 3.32.9 COPY - Copy Library Member

The COPY assembly statement is used to copy previously defined statement sequences from a library to a source program.

Format:

| Name | Operation | Operand |
|---|---|---|
| | COPY | Symbolic name |

The symbolic name is the name of the library member to be copied (see Note 4). It is formed in accordance with the syntax rules for symbolic addresses.

Notes:

- The statements to be copied are inserted immediately following the COPY statement.

- The copied text is interpreted according to the preceding ICTL statement (if used). As this may be unintentional, a warning is issued: Error in COPY.

- The copied text may not contain an ICTL or ISEQ statement.

- The COPY member is initially sought as a TYPE=S member in the same program library as that in which the source program is contained. If the COPY member is not found there, it is sought as a TYPE=M member in the program libraries allocated by means of the link names ALTLIB, ALTLIB2 to ALTLIB5.

  A search for a COPY member is made in the $TSOS.MACROLIB only if the COPY statement is included in a macro from the system macro library.

- If several members exist in the source or macro section of a program library that have the same name but different version numbers, the member having the highest number is used.

- COPY calls may be nested to a depth of 5 and may occur in source or macro instructions.

  Copied statements may contain macro definitions unless they are themselves part of a macro.

- The printing of the copied statements can be permitted with PRINT COPY and suppressed with PRINT NOCOPY (see 3.31.4).

- The name of the member to be copied must not be generated.

Example:

In this example the COPYCOD1 library member (cf. note 4) is inserted in the program to be assembled.

| Name | Operation | Operand |
|---|---|---|
| | COPY | COPYCOD1 |

### 3.32.10 OPSYN - Redefine Mnemonic Operation Code

The OPSYN statement is used to define a mnemonic operation code which is equivalent to another operation code or macro name, or to cancel an operation code for subsequent program statements.

Format:

| Name | Operation code | Operand |
|---|---|---|
| Symbol | OPSYN | Symbol or blank |

The name field symbol is assigned the attributes of the operand field symbol. If the operand field is blank, the name field symbol is deactivated; that is, it will no longer be recognized as a mnemonic operation code. The name field symbol must be built according to the rules for symbolic addresses. It may also be the name of an assembly statement; if so, that statement will be redefined or canceled. If it is canceled, then subsequent statements carrying the same name will be processed as macro instructions.

The operand field symbol may be:

- A machine or assembly instruction
- A statement previously defined by another OPSYN
- A macro name

The OPSYN statement may appear anywhere in the program outside macro definitions. An OPSYN-defined operation code remains valid for all subsequent statements up to the end of the program or until redefinition by another OPSYN statement. Operation codes in statements within source program macros are analyzed according to the OPSYN statements preceding the macro definition. Operation codes in statements within library macros are governed by the OPSYN statements valid at the time of the END statement.

An OPSYN-defined symbol retains its assigned meaning even if the operand field symbol of that OPSYN statement is later redefined by another OPSYN.

An OPSYN-defined symbol retains its assigned meaning even if the operand field symbol of that OPSYN statement is later redefined by another OPSYN.

An OPSYN statement carrying the same symbol in its name and operand fields causes that symbol to be redefined to its initial definition.

Examples of valid OPSYN statements:

```
EOJ     OPSYN     TERM     Redefines the macro TERM
STHH    OPSYN     STH      Redefines the machine instruction STH

*                          The following statement cancels STH as a
                           machine instruction:

STH     OPSYN
ABC     OPSYN     STHH     Introduces 'ABC' additionally and with the
                           same effect as STHH.

*                          The following statement cancels STHH, with ABC
*                          continuing to define the machine instructions STH.

STHH    OPSYN
STH     OPSYN     STH      Readmits the machine instruction STH in its
                           original meaning.
```

### 3.32.11 STACK - Store PRINT, USING Status

The STACK statement is used for storing the USING status (i.e. the base registers and their functions) and the current PRINT parameters.

Format:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | STACK | Sequence of operands PRINT or USING |

The PRINT and USING status may be stored up to four times after one another without a subsequent UNSTK statement.

### 3.32.12 UNSTK - Reset PRINT, USING Status

The UNSTK statement is used for resetting the USING or PRINT status stored with the STACK statement (see STACK).

Format:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | UNSTK | Sequence of operands PRINT or USING |

Every UNSTK statement must be preceded by a corresponding STACK statement.

# 4 MACRO LANGUAGE

## 4.1 GENERAL

The macro language facilities allow frequently used sequences of assembly statements to be written once, in a standardized form - a macro definition - and generated within a program by writing only a single statement, a macro instruction, whenever the particular sequence of statements is desired. This facility simplifies the coding of programs, reduces programming errors and permits the standardization of coding that is used for frequently needed functions.

Conditional assembly and SET instructions provide the programmer with a convenient way to vary the format and number of statements that appear within his program, according to conditions evaluated at assembly time. When used within a macro definition, the conditional assembly statements may be used to vary the statements that are generated for each occurrence of a macro instruction.

## 4.2 THE MACRO INSTRUCTION STATEMENT

A Macro Instruction Statement (hereafter called a macro instruction) is a source program statement which, through the parameters which are written in the operand entry, provides the assembler with information needed to generate source statements from a corresponding macro definition.

Four types of macro instructions may be used:

- Positional macro instruction
- Keyword macro instruction
- Mixed mode macro instruction
- Macro call in alternate statement form

The operands of a positional macro instruction must appear in a fixed order.

Keyword macro instruction operands may be specified in any order.

Mixed mode instruction operands may be either positional or keyword. Each type of operand is discussed in more detail in subsequent sections.

## 4.3 THE MACRO DEFINITION

A macro definition must be available to the assembler before a macro instruction can be processed. The macro definition contains the machine instructions and assembly statements with which the programmer wishes to replace every occurrence of the corresponding macro instruction; it also contains the SET and conditional assembly instructions which control the format and sequence of these statements.

Every macro definition consists of:

1. A header statement.

2. A macro instruction prototype statement.

3. Zero or more model statements, which may be assembly and machine instructions, SET and conditional statements, or MNOTE statements.

4. A trailer statement.

Each class of statements is described in detail in succeeding sections of this publication.

## 4.4 THE MACRO LIBRARY

The same macro definition may be made available to more than one source program by placing the macro definition in a macro library. The macro library is a collection of macro definitions that can be referenced by all assembly language programs in an installation. Once a macro definition has been placed in the macro library, it may be used by writing the corresponding macro instruction in a source program.

The programmer may also keep private macro libraries to be used in conjunction with the system macro library (see 2.4 item 5 and 2.7.2). The procedures for placing macro definitions in the system macro library or in the user's private library are described in the Utilities Manual (MLU) and in the LMS Reference Manual. The search hierarchy of the macros in the libraries is discussed in section 2.7.3.

## 4.5 VARYING THE GENERATED STATEMENTS

Conditional assembly instructions may be used to control the number and sequence of statements that appear in a source program or replace a macro instruction. The formats of generated statements may be varied by the use of variable symbols and the SET instructions. Variable symbols are symbols whose contents may be changed by the programmer or the assembler. Each occurrence of a variable symbol in a model statement of a macro definition or in a source language statement is replaced by the current value assigned to the symbol.



### 4.5.1 Types of Variable Symbols

There are three types of variable symbols:

1. **Symbolic parameters**, which are assigned values by the programmer when he writes a macro instruction.

2. **System variable symbols**, which are assigned values by the assembler each time it processes a macro definition.

3. **SET symbols**, which are assigned values by the programmer inside and outside macro definitions.

Variable symbols may be global or local. Global variable symbols may be used to communicate information between macro definitions and between a macro definition and the source program. Local variable symbols are provided for temporary usage within a single macro definition or in the source program. The values of local variable symbols used in a macro definition are not accessible outside the definition, nor are they preserved between macro definitions. Symbolic parameters and system variable symbols are local variable symbols. SET symbols may be defined as either local or global.

## 4.6 HOW TO PREPARE MACRO DEFINITIONS

A macro definition must be available to the assembler before it can process any macro instruction which references it. The macro definition must appear in the source program before any occurrence of the corresponding macro instruction or must be in the system macro library or one of the user macro libraries.

A macro definition consist of:

1. A macro definition header statement.
2. A macro instruction prototype statement.
3. Zero or more model statements, which may be assembly and machine instructions, conditional assembly and SET instructions, and MNOTE statements.
4. A macro definition trailer statement.

### 4.6.1 MACRO - Macro Definition Header

The macro definition header statement denotes the beginning of a macro definition. It must be the first statement in every macro instruction. Comment lines may precede a header statement in library macros but are ignored.

The format of this statement is:

| Name | Operation | Operand |
|---|---|---|
| Blank | MACRO | Not used |

### 4.6.2 MEND - Macro Definition Trailer

The macro definition trailer statement denotes the end of a macro definition. It must be the last statement in every macro definition.

The format of this statement is:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | MEND | Not used |

## 4.7 MACRO INSTRUCTION PROTOTYPE

The macro instruction prototype statement (hereafter called the prototype statement) specifies the name entry, mnemonic operation code, and the format of all macro instructions that refer to this macro definition. It must be the second statement of every macro definition.

Macro definitions are classified according to the type of prototype statement that is used. The prototype may be written in one of four formats: positional, keyword, mixed mode or alternate statement form.



### 4.7.1 Positional Prototype Statement

Positional macro definitions are used when only a small number of operands will be supplied in the macro instruction or when the number of operands cannot be predetermined (see discussion of &SYSLIST).

The positional prototype statement is written in the following form:

| Name | Operation | Operand |
|---|---|---|
| A symbolic parameter or blank | A symbol (mnemonic operation code | Zero or more symbolic parameters, separated by commas |

The symbolic parameters that appear in the positional prototype statement represent the corresponding fields in the associated macro instruction. A complete description of symbolic parameters follows the discussion of Model Statements.

The symbol in the operation fields is the mnemonic operation code that must appear in the corresponding field of all macro instructions that are to reference this macro definition. The mnemonic operation code of every macro definition in a source program must be unique and may not be the same as any machine or assembly instruction.

The following example illustrates a positional prototype statement, with two symbolic parameters:

| Name | Operation | Operand |
|---|---|---|
| &NAME | MOVE | &TO,&FROM |

### 4.7.2 Keyword Prototype Statement

A keyword macro definition enables the programmer to reduce the number of operands that must be specified in each corresponding macro instruction, to write the operands in any order, and to assign preliminary values to the symbolic parameters that appear in the operand of the prototype statement. These values are used while processing the macro definition, unless the programmer supplies a new value for the symbolic parameter when he writes the macro instruction.

Keyword macro definitions are useful when a macro instruction may have many operands, but normally specifies only a few. In this case, the omitted parameters are assigned the standard values that were written in the prototype statement.

A keyword macro definition is identical to a positional macro definition, except for the operand entry of the prototype statement.

The format of this statement is:

| Name | Operation | Operand |
|---|---|---|
| A symbolic parameter or blank | A symbol (mnemonic operation code) | One or more operands in the form described below, separated by commas |

Each operand consists of a symbolic parameter immediately followed by an equal sign and optionally followed by a value. If the value is not desired or is unnecessary, the equal sign must be immediately followed by the comma that separates the parameter from the next one, or by a space if it is the last parameter.

Anything that may be used as an operand in a positional macro instruction, except variable symbols, may be used as a standard value in the keyword prototype statement.

The following are valid keyword prototype operands:

```
&READER=
&LOOP2=SYMBOL
&S4=F'4096'
```

The following are invalid keyword prototype operands:

```
CARD          (not a symbolic parameter)
&TYPE         (not followed by an = sign)
&TWO  =123    (equal sign does not immediately follow the symbolic
              parameter)

&A= X'01'     (space between equal sign and value)
&B=A&AG1      (variable symbol in standard value)
```

A valid keyword prototype statement with four operands is illustrated in the next example. Note that only two parameters have been given standard values.

| Name | Operation | Operand |
|---|---|---|
| &N | MOVE | &R=2,&A=S,&T=,&F= |

### 4.7.3 Mixed-Mode Prototype Statement

Mixed-mode macro definitions allow the programmer to combine features of keyword and positional macro definitions in the same macro definition.

The format of the mixed-mode prototype statement is:

| Name | Operation | Operand |
|---|---|---|
| A symbolic parameter or blank | A symbol | Two or more operands of the form described below, separated by commas |

The operands must be valid operands of positional and keyword prototype statements. All positional operands must precede the first keyword operand. The rules for forming positional and keyword operands have been discussed earlier in this section.

The following mixed-mode prototype statement illustrates two positional operands and two keyword operands.

| Name | Operation | Operand |
|---|---|---|
| &N | MOVE | &T,&P,&TO=,&F=3 |

### 4.7.4 Alternate Statement Form

The prototype statement may be written in a form different from that used for machine or assembly instructions. The normal form is described in section 3.1.1. of this publication. The alternate form described here allows the programmer to write an operand on each line, and allows the interspersing of operands and comments in the statement.

In the alternate form, as in the normal form, the name and operation entries must appear on the first line of the statement, and at least one blank must follow the operation entry on that line. Both types of statement forms may be used in the same prototype statement.

The rules for using the alternate statement form are:

1. If an operand is followed by a comma and a blank $(40)_{16}$ and the column after the end column contains a non-blank character, the operand entry may be continued on the next line starting in the continue column. More than one operand may appear on the same line.

2. Comments may appear after the blank that indicates the end of an operand, up to and including the end column.

3. If an operand is followed by a blank, it indicates the end of the operand field. Comments may appear up to the end column. If a character other than blank appears in this line in the column following the end column, the comment field is continued in the next line, beginning in the continuation column.

Note

A prototype statement may be written on as many continuation lines as there are operands and associated comments.

The following examples illustrate:

1. the normal statement form

2. the alternate statement form, and

3. the combination of both forms, for a positional prototype statement.

| Name | Operation | Operand | |
|---|---|---|---|
| &NAME1 | OP1 | &OPERND1,&OPERND2,&OPERND3 | X |
| | | Normal Form | |
| &NAME2 | OP2 | &OPERND1, | X |
| | | &OPERND2,&OPERND3 | X |
| | | Alternate Statement Form | |
| &NAME3 | OP3 | &OPERND1, | X |
| | | &OPERND2,&OPERND3,&OPERND4, | X |
| | | &OPERND5 | X |
| | | Combination of both Forms | |

## 4.8 MODEL STATEMENTS

Model statements are the macro definition statements from which the desired sequences of machine and assembly instructions are generated. Any number of model statements may follow the prototype statement. A model statement consists of one to four entries. They are, from left to right, the name, operation, operand, and comments entry.

The name entry may be an ordinary symbol, a sequence symbol, a variable symbol, or a combination of variable symbols and alphanumeric characters which generate an ordinary symbol, depending on the particular statement (see discussion of concatenation, section 4.10). * and .* may be generated by variable symbols in the begin column of a model statement.

The operation entry of a model statement may contain any machine, assembly, or macro instruction mnemonic operation code, except ICTL, ISEQ and MACRO; or it may contain a variable symbol. Variable symbols may not be used to generate the following operation codes:

ICTL, ISEQ, REPRO, START, COM, MACRO, MEND, MEXIT, MNOTE, LCLA, LCLB, LCLC, GBLA, GBLB, GBLC, SETA, SETC, SET, AIF, AIFB, AGO, AGOB, ANOP and ACTR.

Variable symbols may not be used to generate macro instruction operation codes, except in special cases (see MCALL statement, 4.13.9).

Variable symbols may be used in assembly language statements outside macro definitions under the same restrictions as given above for macro model statements. Note, however, that symbolic parameters may not be used outside macro definitions and that the presence of variable symbols in a statement may inhibit complete attribute processing (see discussion of "Attributes", section 4.15).

The operand entry may contain ordinary symbols, variable symbols, and other assembly language elements in accordance with the rules for paired apostrophes, ampersands and blanks (see section 4.13.1, "Macro Instruction Operands"). After all variable symbol substitution has been completed, the resulting operand field must not extend over more than two continuation lines.

The comments entry may contain any combination of characters. Variable symbols that occur within the comments field are not replaced. Similarly, the line following a REPRO statement may not be generated by variable symbols.

A single variable symbol may not be used to generate more than one entry, but more than one variable symbol may appear in the same model statement.

## 4.9 SYMBOLIC PARAMETERS

A symbolic parameter is a type of variable symbol that is assigned a value by the programmer each time he writes a macro instruction or, for a keyword or mixed mode macro definition, the macro prototype statement. Symbolic parameters may be used wherever variable symbols are permitted unless otherwise stated. By changing the values assigned to symbolic parameters, the programmer can vary the statements that are generated or each occurrence of a macro instruction. The value of a symbolic parameter may also be assigned or changed using a SETC statement.

A symbolic parameter consists of an & followed by 1 to 7 alphanumeric characters (letters and digits), the first of which must be a letter.

The programmer should not use any symbolic parameters which begin with &SYS (see 4.14.1 last paragraph).

The same variable symbol may not be redefined as a symbolic parameter and as a SET symbol in the same macro definition.

The same symbolic parameter may be used in two or more macro definitions within a single program.

The following are valid symbolic parameters:

```
&READER        &LOOP2
&A             &S4
&X4F2
```

The following are invalid symbolic parameters:

```
&256B          (first character after the ampersand is not a letter)
&AREA2456      (more than seven characters after the ampersand)
&BCD(34)       (contains a special character other than the initial ampersand)
&IN_AREA       (contains a special character other than the initial ampersand)
&SYSTEM        (is the system variable symbol &SYSTEM)
```

Any symbolic parameters that are used within a macro definition must appear in the macro definition prototype. Parameters in model statements are replaced by the characters that currently correspond to them.

The following keyword macro definition illustrates the use of symbolic parameters in model statements. Note that one symbolic parameter has been given a value in the prototype. This value will be used unless a new one is assigned in the macro instruction.

| | Name | Operation | Operand |
|---|---|---|---|
| Header | | MACRO | |
| Prototype | &NAME | MOVE | &TO=,&FROM=WORK |
| Model | &NAME | ST | 2,SAVE |
| Model | | L | 2,&FROM |
| Model | | ST | 2,&TO |
| Model | | L | 2,SAVE |
| Trailer | | MEND | |

In the following example, the symbols HERE, FIELDA, and FIELDB are written in the MOVE macro instruction to correspond to the symbolic parameters &NAME , &TO, and &FROM, respectively, of the MOVE prototype statement.

| Name | Operation | Operand |
|---|---|---|
| HERE | MOVE | TO=FIELDA,FROM=FIELDB |

The following assembly language statements would be generated by the preceding macro instruction.

| Name | Operation | Operand |
|---|---|---|
| HERE | ST | 2,SAVE |
| | L | 2,FIELDB |
| | ST | 2,FIELDA |
| | L | 2,SAVE |

The generated statements can be varied by changing the values of the symbolic parameters supplied in the macro instruction, as illustrated in the following example.

| Name | Operation | Operand |
|---|---|---|
| LABEL | MOVE | TO=IN,FROM=OUT |
| LABEL | ST | 2,SAVE |
| | L | 2,OUT |
| | ST | 2,IN |
| | L | 2,SAVE |

A symbolic parameter will not be replaced if it appears in the comments field of a model statement.

## 4.10 CONCATENATION

If a variable symbol in model statement is immediately preceded or followed by other characters or variable symbols, the characters that correspond to the variable symbol are combined, in the generated statement, with the other characters or with the characters that correspond to the other variable symbols. This process is called **concatenation**.

The macro definition, macro instruction, and generated statements that follow illustrate the rules for concatenating symbolic parameters with other characters or other symbolic parameters.
Additional concatenation techniques are described in the discussion of "Evaluation of Character Expressions", section 4.17.2.3.

| | Name | Operation | Operand |
|---|---|---|---|
| Header | | MACRO | |
| Prototype | &NAME | MOVE | &TY,&P,&TO,&FROM |
| Model | &NAME | ST&TY | 2,SAVEAREA |
| Model | | L&TY | 2,&P&FROM |
| Model | | ST&TY | 2,&P&TO |
| Model | | L&TY | 2,SAVEAREA |
| Trailer | | MEND | 2,SAVEAREA |
| Macro | HERE | MOVE | D,FIELD,A,B |
| Generated | HERE | STD | 2,SAVEAREA |
| Generated | | LD | 2,FIELDB |
| Generated | | STD | 2,FIELDA |
| Generated | | LD | 2,SAVEAREA |

The symbolic parameter &TY is used in each of the four model statements to vary the mnemonic operation code of each of the generated statements. The character D in the macro instruction corresponds to symbolic parameter &TY. Since &TY is preceded by other characters (i.e., ST and L) in the model statements, the character that corresponds to &TY (i.e., D) is concatenated with the other characters to form the operation fields of the generated statements.

The symbolic parameters &P, &TO, and &FROM are used in two of the model statements to vary part of the operand fields of the corresponding generated statements. The character FIELD, A, and B correspond to the symbolic parameters &P, &TO, and &FROM, respectively. Since &P is followed by &FROM in the second model statement, the characters that correspond to them (i.e., FIELD and B) are concatenated to form part of the operand field of the second generated statement. Similarly, FIELD and A are concatenated to form part of the operand field of the third generated statement.

If the programmer wishes to concatenate a symbolic parameter with a letter, digit, left parenthesis, or period following the symbolic parameter, he must immediately follow the symbolic parameter with a period. A period is optional if the symbolic parameter is to be concatenated with another symbolic parameter, or a special character other than a left parenthesis or another period that follows it.

If a symbolic parameter is immediately followed by a period, then the symbolic parameter and the period are replaced by the characters that correspond to the symbolic parameter. A period that immediately follows a symbolic parameter does not appear in the generated statement.

In the following examples, assume the symbolic parameter &PARAM has a value of A.

| Model Statement Representation | Generated Characters |
|---|---|
| &PARAM.(BC) | A(BC) |
| &PARAM..BC | A.BC |
| &PARAM.BC | ABC |
| &PARAM.2BC | A2BC |
| &PARAM,.2B | A,.2B |
| BC&PARAM | BCA |
| BC,&PARAM | BC,A |
| B2&PARAM | B2A |
| &PARAM.&PARAM | AA |
| &PARAM&PARAM | AA |
| &PARAM..&PARAM | A.A |



The following macro definition, macro instruction, and generated statements illustrate these rules.

| | Name | Operation | Operand |
|---|---|---|---|
| Header | | MACRO | |
| Prototype | &NAME | MOVE | &P,&S,&R1,&R2 |
| Model | &NAME | ST | &R1,&S.(&R2) |
| Model | | L | &R1,&P.B |
| Model | | ST | &R1,&P.A |
| Model | | L | &R1,&S.(&R2) |
| Trailer | | MEND | |
| Macro | HERE | MOVE | FIELD,SAVE,2,4 |
| Generated | HERE | ST | 2,SAVE(4) |
| Generated | | L | 2,FIELDB |
| Generated | | ST | 2,FIELDA |
| Generated | | L | 2,SAVE(4) |

The symbolic parameter &P is used in the second and third model statements to vary part of the operand field of each of the corresponding generated statements. The characters FIELD of the macro instruction correspond to &P. Since &P is to be concatenated with a letter (i.e., B and A) in each of the statements, a period immediately follows &P in each of the model statements. The period does not appear in the generated statements.

Similarly, symbolic parameter &S is used in the first and fourth model statements. In the operand of the corresponding statements &S is followed by a period, because it is to be concatenated with a left parenthesis. The period does not appear in the generated statements.

## 4.11 MNOTE STATEMENT

An MNOTE statement may be used as a model statement in a macro definition or in the source program.
It may be used to generate a variable message between source statements or in a macro expansion.

The format of the MNOTE statement is:

| Name | Operation | Operand |
|---|---|---|
| A sequence symbol or blank | MNOTE | [{nnn / *},]'character string'<br>0 ≤ nnn ≤ 255 |

The error code may be generated by a variable symbol.

The error code (nnn) may contain a maximum of 3 digits; leading zeros are permitted.

If the error code is omitted, 0 is assumed.

If the error code is not an integer or is omitted, the statement will be treated as an error message and listed like other invalid model statements even if PRINT NOGEN is specified.

The following relationship exists between the error codes of the MNOTES and the error classes of the assembler flags:

| | | | |
|---|---|---|---|
| | Error code = 0 | Error class 0 | Warning |
| 0 < | Error code ≤ 150 | Error class 1 | Error |
| 150 < | Error code ≤ 254 | Error class 2 | Severe error |
| | Error code = 255 | Error class 3 | Fatal error |

It is thus possible to terminate the assembly prematurely by generating a MNOTE with error code 255. In this case the assembler simply outputs an ERROR POOL.

If the error code is an asterisk, the operand field of the MNOTE statement will be listed as a comments statement, in the generated code, even if PRINT NOGEN is specified. The MNOTE operation code and the apostrophes enclosing the message will not appear.

At the end of the assembly listing the diagnostic listing is followed by a listing of the MNOTES in accordance with the error codes. Cross-references are specified for each error code.

MNOTE *'...' is assigned no error weight, i.e., the end message reports HIGHEST ERROR WEIGHT: -, unless other warnings or errors occurred.

Variable symbols in the operand field of either form of the MNOTE statement will be replaced by their current values. In accordance with the rules for model statement operand fields, two apostrophes must be used to generate a single apostrophe. If ampersands that are not part of a variable symbol are used, then two ampersands must be written. In contrast to assembly language they are used as two separate characters. An inquiry for an ampersand may be entered by means of a partial character string ('&&'(1,1)).

## 4.12 COMMENTS STATEMENTS

A model statement may be a comments statement. A comments statement consists of an asterisk in the begin column, followed by comments. The comments statement is used by the assembler to generate an assembly language comments statement, just as other model statements are used by the assembler to generate assembly language statements.

The programmer may also write comments statements in a macro definition which are not to be generated. They are used for documentation purposes only. These statements must have a period in the begin column, immediately followed by an asterisk and the comments.

If in a library macro a header statement is preceded by comment lines (asterisk in first column) such lines are ignored.

The first statement in the following example will be used by the assembler to generate a comments statement; the second statement will not.

| Name | Operation | Operand |
|---|---|---|
| * THIS STATEMENT WILL BE GENERATED | | |
| .* THIS ONE WILL NOT BE GENERATED | | |

**Generation of comments cards**

The macro language makes it possible to generate comments from statements and instructions, thus deactivating their function. It is however not possible to generate any kind of comments.

Statements which are allowed to have a symbol in the name field are written in the source program with a variable SETC symbol as name. If this variable is assigned an '*' by generation in the macro processor, then the corresponding statement is made ineffective and is logged as comments.

The logging of the generated comments is suppressed if the variable SETC symbol is assigned the identifier of macro comments '.*'.

Note:

It is true that the effect of macro instructions will at the same time be nullified, but on account of the assembler structure it is not possible to prevent them from being read in.

## 4.13 MACRO INSTRUCTIONS

A macro instruction is an assembly language statement that represents a variable sequence of statements which are written in a corresponding macro definition. Each occurrence of a macro instruction will be replaced by selected statements from the macro definition. The form of the generated statements can be controlled by varying the operands of the macro instruction.

There are four types of macro instructions corresponding to the four macro definition types. They are: positional, keyword, mixed mode and alternate format (see section 4.13.2.

### 4.13.1 Macro Instruction Operands

Any combination of up to 127 characters may be used as a macro instruction operand provided that the following rules are observed. These rules apply to the format of macro instruction operands before substitution for variable symbols.

**Paired Apostrophes:** An operand may contain apostrophes, provided they occur in pairs. Two apostrophes must be used to represent a single apostrophe that appears between paired apostrophes.

In the example below, the first and fourth apostrophes, and the fifth and sixth apostrophes are paired apostrophes.

'A''B'C'D'

An apostrophe immediately followed by a letter and immediately preceded by the letter L (when L is preceded by any character other than an &) is not considered in determining paired apostrophes. For instance, L'SYMBOL is a valid operand.

**Paired Parentheses:** An operand must contain an equal number of left and right parentheses. The nth left parenthesis must appear to the left of the nth right parenthesis. A left parenthesis and a following right parenthesis without any other intervening parentheses, form a set of paired parentheses. If there is more than one pair, each additional pair is determined by removing any pairs already recognized and reapplying the above rule for paired parentheses. In the following example, the first and fourth, the second and third, and the fifth and sixth parentheses are paired parentheses.

Q(A(B),C)D(E)

If an operand begins with a left parenthesis, it must terminate with a matching right parenthesis and will be treated as a sublist (see Operand Sublists).

Any remainder on the right of the right parenthesis is ignored (this applies particularly to arithmetic expressions) and the line is flagged.

A parenthesis that appears between paired apostrophes is not considered in determining paired parentheses. For instance, in the following example the middle parenthesis is not considered.

```
(')')
```

**Equal Signs:** An equal sign can only occur as the first character in an operand or between paired apostrophes or paired parentheses. The following examples illustrate these rules.

```
=F'32'          'C=D'          E(F=G)
```



**Ampersands (&):** Each sequence of consecutive ampersands must contain an even number, except when the ampersand is part of a variable symbol. The following example illustrates this rule.

Example: **&&123&&&&**

Commas: A comma indicates the end of an operand, unless it is placed between paired apostrophes or paired parentheses. The following examples illustrate this rule.

Example: (A,B)C','

Blanks: A blank indicates the end of the operand entry.

Exceptions: It meets the statement format requirements (see 3.1.1.3) or it is placed between paired apostrophes. The following example illustrates this rule.

Example: 'A B C'

The following are valid macro instruction operands:

```
SYMBOL          A+2
123             (TO(8),FROM)
X'189A'         0(2,3)
*               =F'4096'
L'NAME          AB&&9
'TEN = 10'      'PARENTHESIS IS)'
'COMMA IS,'     'APOSTROPHE IS'''
```

The following are invalid macro instruction operands:

```
W'NAME          odd number of apostrophes
5A)B            number of left parentheses does not equal number of
                right parentheses
(15 B)          blank not placed between paired apostrophes
'ONE' IS'1'     blank not placed between paired apostrophes
(A,B A          operand begins with a left parenthesis, but does not ter-
                minate with a right parenthesis
```

**Variable Symbols:** Variable symbols may appear in the operand field of a macro instruction. Only SET variable symbols may be used in an outer macro instruction, while symbolic parameters, system variable symbols and SET symbols may be used in an inner macro instruction (see discussion of "Inner Macro Instructions", section 4.13.7). Variable symbols will be replaced by the values currently assigned to them as the macro instruction operands are processed.

**Operand Sublists:** An operand of a macro instruction may be a sublist. A sublist consists of one or more operands separated by commas and enclosed in paired parentheses. The entire sublist, including the parentheses is considered to be one macro instruction operand. The limit of 127 characters per operand applies to the entire sublist, including all suboperands, commas, and parentheses (cf Appendix A9, item 12).

Sublists provide the programmer with a convenient way of referring to:

1. a collection of operands as a single operand or

2. a single operand within a collection of operands.

Sublists may be nested to any level. The value and attributes of any suboperand of an outer or inner level sublist are accessible under the restrictions discussed in "Attributes", section 4.15. Other suboperands are handled in the same way as omitted positional operands.

If &P1 is a symbolic parameter in a prototype statement, and the corresponding operand of a macro instruction is a sublist, then &P1(n) may be used in a model statement to refer to the nth operand of the sublist, where n may be any arithmetic expression allowed in a SETA instruction (see section 4.17.1 for details on the SETA instruction). If &P1(n) is, in turn, a sublist, then &P1(n,m) may be used to refer to the nth sub-operand of the inner sublist.

Indexing may be extended to refer to any element within any level of sublist.

The following examples illustrate the use of indexing to refer to statements. &P is a symbolic parameter and has a value (A,((B,C),)).

| Model Statement Representation | Generated Characters |
|---|---|
| &P(1) | A |
| &P(2,1) | (B,C) |
| &P(2,2) | null |
| &P(1,1) | A |
| &P | (A,((B,C),)) |
| &P(2,3) | null |
| &P(1,2) | null |
| &P(2,1,2) | C |

Note

that if the sublist notation is used, and the operand is not a sublist (or its equivalent for an inner sublist), &P(1) refers to the entire operand, and &P(m), where m ≥ 2, refers to a null character value.

The following macro definition, macro instruction and generated statements further illustrate the use of sublists.

| | Name | Operation | Operand |
|---|---|---|---|
| Header | | MACRO | |
| Prototype | | ADDNUM | &NUM,&REG,&AREA |
| Model | | L | &REG,&NUM(1) |
| Model | | A | &REG,&NUM(2) |
| Model | | A | &REG,&NUM(3) |
| Model | | ST | &REG,&AREA |
| Trailer | | MEND | |
| Macro | | ADDNUM | (A,B,C),6,SUM |
| Generated | | L | 6,A |
| Generated | | A | 6,B |
| Generated | | A | 6,C |
| Generated | | ST | 6,SUM |

The operand of the macro instruction that corresponds to symbolic parameter &NUM is a sublist. One of the operands in the sublist is referred to in the operand entry of three of the model statements. For example, &NUM(1) refers to the first operand in the sublist corresponding to symbolic parameter &NUM. The first operand of the sublist is A. Therefore, A replaces &NUM(1) to form part of the generated statement.

Note

When referring to an operand in a sublist, the left parenthesis of the sublist notation must immediately follow the last character of the symbolic parameter, e.g., &NUM(1). A period should not be placed between the left parenthesis and the last character of the symbolic parameter. A period must be used between these two characters only when the programmer wants to concatenate the left parenthesis with the characters that the symbolic parameter represents. The following example shows what would be generated if a period appeared between the left parenthesis and the last character of the symbolic parameter in the first model statement of the above example.

| | Name | Operation | Operand |
|---|---|---|---|
| Prototype | | ADDNUM | &NUM,&REG,&AREA |
| Model | | L | &REG,&NUM.(1) |
| Macro | | ADDNUM | (A,B,C),6,SUM |
| Generated | | L | 6,(A,B,C)(1) |

The symbolic parameter &NUM is used in the operand entry of the model statement. The characters (A,B,C) of the macro instruction correspond to &NUM. Since &NUM is immediately followed by a period, &NUM and the period are replaced by (A,B,C). The period does not appear in the generated statement. The resulting generated statement is an invalid assembly language statement.

### 4.13.2 Positional Macro Instruction

The format of a positional macro instruction is:

| Name | Operation | Operand |
|---|---|---|
| A symbol, sequence symbol, or blank | Mnemonic operation code | Zero or more operands separated by commas |

The name entry of the macro instruction may contain a symbol. The symbol will not be defined in the generation process unless a symbolic parameter appears in the name entry of the prototype and the same parameter appears in the name entry of a generated model statement.

The operation entry contains the mnemonic operation code of the macro instruction. It must be the same as the mnemonic operation code of a macro definition prototype statement in the source program or in the system or user macro library.

The macro definition with the same mnemonic operation code in the prototype statement is used by the assembler to process the macro instruction. If a macro definition in the source program and one in a macro library have the same mnemonic operation code, the source program definition will be used. If a macro definition in a user's macro library has the same mnemonic operation code as in the system macro library, the user's macro definition will be used.

The position and order of the operands in a positional macro instruction are determined by position and order of the symbolic parameters in the operand of the corresponding prototype statement. The number of operands that appear in the positional macro instruction operand may be greater than the number of symbolic parameters that are written in the prototype statement. In this case, the &SYSLIST facility may be used to reference the additional operands (see section 4.14.1.3).

### 4.13.3 Omitted Operands

If an operand that appears in the positional prototype statement is omitted from the macro instruction, then the comma that would have separated it from the next operand must be present. If the last n positional operands are omitted, the last n commas may be omitted.

Omitting an operand from a macro instruction causes the corresponding symbolic parameter to be assigned the null string (not a character) as the character value (as operand in the SETC statements or in character relations). As an arithmetic value it has the null value (in arithmetic relations or as the operand of an SETA statement).



The rules for writing macro instruction operands and examples of positional macro instruction are given below.

### 4.13.4 Keyword Macro Instruction

A keyword macro definition is used by writing a keyword macro instruction.

The format of this type of macro instruction is:

| Name | Operation | Operand |
|---|---|---|
| A symbol, sequence symbol, or blank | Mnemonic Operation Code | Zero or more operands in the form described below, separated by commas. |

Each operand consists of a keyword beginning with a letter and followed immediately by an equal sign and an optional value. A keyword is a symbolic parameter, written without the initial ampersand, that appears in the corresponding macro definition prototype statement. Any keywords used in the macro instruction must appear in the associated macro definition prototype.

The rules for specifying the value of a keyword are identical to the rules governing positional macro instruction operands. Anything that may be used as an operand in a positional macro instruction may be used as a value in a keyword macro instruction. Nested keywords are not permitted.

Keyword operands may be written in any order. Because of this feature, it is not necessary to supply the comma that separates operands whenever an operand is omitted. An operand which is omitted from the macro instruction and was not given a standard value in the prototype statement will be assigned a null character value.

Symbolic parameters used in keyword macro prototypes assume the values assigned to them in the macro instruction.

The rules under which the assembler replaces symbolic parameters in model statements with the values is described in Positional Macro Instructions.

The following examples illustrate a keyword macro definition, a keyword macro instruction and the resulting generated statements.

Statement 1 assigns the values 2 and S to the symbolic parameters &R and &A, respectively. Statement 6 assigns the values FA, FB and THERE to the keywords T, F, and A, respectively. The symbol HERE is used in the name entry of statement 6.

Since a symbolic parameter (&N) appears in the name entry of the prototype statement (statement 1), and the corresponding characters (HERE) of the macro instruction (statement 6) are a symbol, &N is replaced by HERE in statement 2.

| Name | Operation | Operand |
|---|---|---|
| | MACRO | |
| 1 &N | MOVE | &R=2,&A=S,&T=,&F= |
| 2 &N | ST | &R,&A |
| 3 | L | &R,&F |
| 4 | ST | &R,&T |
| 5 | L | &R,&A |
| | MEND | |
| 6 HERE | MOVE | T=FA,F=FB,A=THERE |
| HERE | ST | 2,THERE |
| | L | 2,FB |
| | ST | 2,FA |
| | L | 2,THERE |

Since &T appears in the operand of statement 1, and statement 6 contains the keyword (T) that corresponds to &T, the value assigned to T (FA) replaces &T in statement 4. Similarly, FB and THERE replace &F and &A in statement 3 and in statements 2 and 5, respectively. Note that the value assigned to &A in statement 6 is used instead of the value assigned to &A in statement 1.

Since &R appears in the operand of statement 1, and statement 6 does not contain a corresponding keyword, the value assigned to &R in the prototype statement replaces &R in statements 2, 3, 4 and 5.

### 4.13.5 Mixed Mode Macro Instruction

The mixed mode macro instruction combines the features of positional and keyword macro instruction. The format of a mixed mode macro instruction is:

| Name | Operation | Operand |
|---|---|---|
| A symbol, sequence symbol or blank | Mnemonic Operation Code | Two or more operands of the form described below, separated by commas |

The operand field is a combination of positional and keyword macro instruction operands. All positional operands must precede the first keyword operand. The positional operands follow the rules for positional macro instructions and the keyword operands follow the rules for keyword macro instructions.

The following mixed mode macro definition, mixed mode macro instruction and generated statements illustrate these facilities.

| Name | Operation | Operand |
|---|---|---|
| | MACRO | |
| &N | MOVE | &T,&P,&Q,&R,&TO=,&F= |
| &N | ST&T | &R,SAVE&Q |
| | L&T | &R,&P&F |
| | ST&T | &R,&P&TO |
| | L&T | &R,SAVE |
| | MEND | |
| HERE | MOVE | H,,,2,F=FB,TO=FA |
| HERE | STH | 2,SAVE |
| | LH | 2,FB |
| | STH | 2,FA |
| | LH | 2,SAVE |

### 4.13.6 Statement Form

Note that the positional operands are in the same order in the macro instruction as the symbolic parameters appear in the prototype statement. The omission of the second and third positional parameters is treated in the same manner as for a positional macro definition. The keyword operands are written in a different order than appeared in the prototype statement. The macro instruction may also conform to the alternate instruction format as discussed in section 4.7.4.

### 4.13.7 Inner Macro Instructions

Macro instructions may be written in the same alternate format that was discussed in section 4.7, "Macro Instruction Prototype". As many continuation lines as are needed to contain all operand entries and associated comments may be used.

A positional, keyword or mixed mode macro instruction may be used as a model line in any macro definition. Macro instructions used in this way are called inner macro instructions, in contrast to outer macro instructions, which appear outside macro definitions.

If a macro definition is contained in the system macro library, the search for 'inner macro' is made normally in that library only.

Any symbolic parameters used in an inner macro instruction are replaced by the corresponding operands of the outer macro instruction. Any ordinary symbols used as operands in an inner macro instruction will not be assigned attributes. The macro definition corresponding to an inner macro instruction is used to generate the statements that replace the inner macro instruction.

The ADDNUM macro instruction of the previous example is used as an inner macro instruction in the following example.

The inner macro instruction contains two symbolic parameters, &S and &T. The characters (X,Y,Z) and J of the macro instruction correspond to &S and &T, respectively. Therefore, these characters replace the symbolic parameters in the operand entry of the inner macro instruction.

The assembler then uses the macro definition that corresponds to the inner macro instruction. The fourth through seventh generated statements have been generated for the inner macro instruction.

| | Name | Operation | Operand |
|---|---|---|---|
| Header | | MACRO | |
| Prototype | | COMP | &R1,&R2,&S,&T,&U |
| Model | | SR | &R1,&R2 |
| Model | | C | &R1,&T |
| Model | | BNE | &U |
| Inner | | ADDNUM | &S,12,&T |
| Model | &U | A | &R1,&T |
| | | MEND | |
| | | MACRO | |
| | | ADDNUM | &NUM,&REG,&AREA |
| | | L | &REG,&NUM(1) |
| | | A | &REG,&NUM(2) |
| | | A | &REG,&NUM(3) |
| | | ST | &REG,&AREA |
| Outer | K | COMP | 10,11,(X,Y,Z),J,K |
| Generated | | SR | 10,11 |
| | | C | 10,J |
| | | BNE | K |
| Generated | | L | 12,X |
| Generated | | A | 12,Y |
| Generated | | A | 12,Z |
| Generated | | ST | 12,J |
| Generated | K | A | 10,J |

Note

An ampersand that is part of a symbolic parameter is not considered in determining whether a macro instruction operand contains an even number of consecutive ampersands.

The following may not be used as operands of an inner macro:

1. A sublist, if it is generated in the outer macro via a SETC statement.

2. A keyword operand (likewise).

3. Several operands separated by commas.

### 4.13.8 Levels of Macro Instructions

A macro definition that corresponds to an outer macro instruction may contain any number of inner macro instructions. The outer macro instruction is called a first level macro instruction. Each of the inner macro instructions is called a second level macro instruction.
The macro definition that corresponds to a second level macro instruction may contain any number of inner macro instructions. These macro instructions are called third level macro instructions, etc.

The maximum number of levels of macro instructions that may be used is 255. The actual number depends on the complexity of the macro definition and the amount of storage available. If the limit is exceeded the macro expansion is terminated (see also Appendix A.9).

Only the numbers of the last two levels used appear in the Assembly Listing.

### 4.13.9 Generated Macro Names in Macro Instructions

The OPCODE field of a statement may contain a symbolic parameter or a SETC symbol. The assigned current value is therefore a character string. The assembler can identify this character string as a macro name only if it is defined in a MCALL statement. In this case, the statement with the generated character string is interpreted as a macro call.

### 4.13.9.1 Define Macro Name (MCALL)

This statement is used to define all macro names which are not directly called in the program but which are **generated** in a macro instruction.

Format:

| Name | Operation | Operand |
|---|---|---|
| Not used | MCALL | One or more macro names separated by commas |



Notes

- Any number of MCALL statements may occur at any position inside or outside macro definitions.
- MCALL statements may contain a maximum of 400 macro names.

Example:

MACRO1 and MACRO2 are two macros in a library. They are to be generated as inner macro instructions in MACRO3.

```
  .
  .
  .
MACRO <—— Definition of the outer macro
MACRO3 &P
  .
  .
  .
MCALL MACRO1,MACRO2 <—— MCALL statement
  .
  .
  .
&P    A,B <—— Symbolic macro instruction
  .
  .
  .
MEND
  .
  .
  .
MACRO3 MACRO1 <—— Outer macro instruction
  .
  .
  .
MACRO1 A,B <—— Generated macro instruction
```

## 4.14 VARIABLE SYMBOLS AND SET INSTRUCTIONS

Variable symbols may appear in macro definition model lines and in most assembly language source program statements outside macro definitions. Where the use of a term or an instruction outside macro definitions differs from its use within macro definitions, the difference will be described in the subsequent text.

Nine instructions are provided to allow the programmer to define and use variable symbols, and in particular, the SET symbols. They are:

| | | |
|---|---|---|
| LCLA | GBLA | SET(A) |
| LCLB | GBLB | SET(B) |
| LCLC | GBLC | SET(C) |

The LCLA, LCLB and LCLC instructions are used to define certain SET symbols as local variable symbols, and to assign them initial values. The GBLA, GBLB and GBLC instructions are used to define certain SET symbols as global variable symbols, and assign them initial values.

The SET(A), SET(B) and SET(C) instructions are used to assign arithmetic, binary, and character values, respectively, to SET symbols. The SETB instruction is described after the SETA and SETC instructions.

### 4.14.1 Variable Symbols

Variable symbols may be used in assembly language statements to vary the format of generated coding. Variable symbols may also be used in conditional SET instructions to perform the arithmetic calculations, character manipulations, and logical operations which ultimately determine the sequence in which statements are assembled into a program.
Variable symbols include SET symbols, symbolic parameters, and system variable symbols.
System variable symbols (&SYSLIST, &SYSECT, &SYSNDX, %SYSVERM), like symbolic parameters, are local to a single macro definition. They are assigned values automatically by the assembler and can only be changed by the programmer by writing another macro instruction.
SET symbols are variable symbols which may appear inside and outside macro definitions. SET symbols differ from symbolic parameters and system variable symbols in three ways:

1. where they can be used in an assembly language program,
2. how they are assigned values, and
3. how the values assigned to them can be changed.

The differences between the three types of variable symbols and their uses are described in detail below.

The system variable symbols &SYSNDX, &SYSECT and &SYSLIST may be used in the name, operation and operand fields of macro definition model statements, but not in statements outside macro definitions. Furthermore, they may not be used as symbolic parameters or SET symbols, nor may they be assigned values by the SET instructions. The other system variable symbols are &SYSVERS, &SYSDATE, &SYSTEM, &SYSTIME, &SYSPARM and &SYSVERM.

When new system symbols are introduced, these are again prefixed with &SYS. To avoid any collisions the following symbols must not begin with the character string &SYS:

- Symbolic parameters
- Variable symbols
- SET symbols



#### 4.14.1.1 &SYSNDX

The system variable symbol &SYSNDX represents a four-digit number that is assigned when any macro instruction is processed. This number is set to 0001 for the first macro instruction and is incremented by one for each subsequent inner or outer macro instruction that is assembled.

Throughout the processing of one macro definition, the value of &SYSNDX may be considered a constant, independent of any inner macro instructions in that definition. Thus, &SYSNDX may be used to create unique names for statements generated from the same model statement, as illustrated in the examples below.

If &SYSNDX is used in a generated model statement, SETC or MNOTE instruction, or a character relation in a SETB or AIF instruction, the value substituted for &SYSNDX is the character representation of the four-digit number assigned to the current macro instruction being processed.
If &SYSNDX appears in arithmetic expressions (e.g., in the operand of a SETA instruction), the value used for &SYSNDX is an arithmetic value.

The following macro definitions and expansions illustrate the use of &SYSNDX. It is assumed that the macro instruction OUTER is the 106th macro instruction processed by the assembler.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | | INNER | &PARAM |
| 1 | A&SYSNDX | SR | 2,5 |
| | | CR | 2,5 |
| | | BE | B&PARAM |
| 2 | | B | A&SYSNDX |
| | | MEND | |
| | | MACRO | |
| | &NAME | OUTER | &PARAM |
| | &NAME | SR | 2,4 |
| 3 | B&SYSNDX | AR | 2,6 |
| 4 | | INNER | &PARAM |
| | B&PARAM | S | 2,=F'1000' |
| 5 | A&SYSNDX | ST | 2,WORD |
| | | MEND | |
| 6 | ALPHA | OUTER | 0300 |
| | ALPHA | SR | 2,4 |
| 7 | B0106 | AR | 2,6 |
| | | INNER | 0300 |
| 8 | A0107 | SR | 2,5 |
| | | CR | 2,5 |
| | | BE | B0300 |
| | | B | A0107 |
| 9 | B0300 | S | 2,=F'1000' |
| 10 | A0106 | ST | 2,WORD |
| 11 | BETA | OUTER | 0400 |
| | BETA | SR | 2,4 |
| | B0108 | AR | 2,6 |
| | | INNER | 0400 |
| | A0109 | SR | 2,5 |
| | | CR | 2,5 |
| | | BE | B0400 |
| | | B | A0109 |
| | B0400 | S | 2,=F'1000' |
| | A0108 | ST | 2,WORD |

Statement 6 is the 106th macro instruction to be processed in this assembly. Therefore, &SYSNDX is assigned the value 0106 throughout the processing of the macro definition for OUTER. This number replaces &SYSNDX in statement 3 and 5, generating the unique names B0106 and A0106.

Statement 4 is the 107th macro instruction processed. Therefore, &SYSNDX is assigned the value 0107 throughout the processing of the INNER macro instruction. 0107 replaces &SYSNDX in statement 1 and 2, generating the unique name A0107.

When the macro instruction OUTER is written again, in statement 11, the same model lines are processed, but because &SYSNDX now has a value of 0108, the symbols which name statements 4 and 5 will be unique for each occurrence of the same macro instruction. Note also that the presence of the inner macro instruction INNER does not affect the value of &SYSNDX in the outer macro definition.

### 4.14.1.2 &SYSECT

The system variable symbol &SYSECT may be used to represent the name of the control section in which a macro instruction appears. For each inner and outer macro instruction processed by the assembler, &SYSECT is assigned a value that is the name of the control section in which the macro instruction appears. &SYSECT may be used in model statements and in conditional assembly instructions.

When &SYSECT is used in a macro definition, the value substituted for &SYSECT is the name of the last CSECT, DSECT, or START statement that occurs before the macro instruction. If no named CSECT, DSECT, or START statements occur before a macro instruction, &SYSECT is assigned a null character value for that macro instruction, i.e. no character is assigned to it.

CSECT or DSECT statements processed in a macro definition affect the value of &SYSECT for any subsequent inner macro instructions in that definition, and for any other outer and inner macro instructions.

Throughout the processing of a macro definition, the value of &SYSECT may be considered a constant for this macro level, independent of any CSECT or DSECT statements or inner macro instructions in that definition. &SYSECT will take on the name of the last CSECT, DSECT, or START statement regardless of whether or not that statement is correct.

The next example illustrates these rules.

Statement 8 is the last CSECT, DSECT, or START statement processed before statement 9 is processed. Therefore, &SYSECT is assigned the value MAINPROG for macro instruction OUTER1 in statement 9. MAINPROG is substituted for &SYSECT when it appears in statement 6.

Statement 3 is the last CSECT, DSECT or START statement processed before statement 4 is processed. Therefore, &SYSECT is assigned the value CSOUT1 for macro instruction INNER in the statement 4. CSOUT1 is substituted for &SYSECT when it appears in statement 2.

Statement 1 is used to generate a CSECT statement for statement 4. This is the last CSECT, DSECT, or START statement that appears before statement 5. Therefore, &SYSECT is assigned the value INA for macro instruction INNER in statement 5. INA is substituted for &SYSECT when it appears in statement 2.

Statement 1 is used to generate a CSECT statement for statement 5. This is the last CSECT, DSECT, or START statement that appears before statement 10. Therefore, &SYSECT is assigned the value INB for macro instruction OUTER2 in statement 10. INB is substituted for &SYSECT when it appears in statement 7.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | | INNER | &INCSECT |
| 1 | &INCSECT | CSECT | |
| 2 | | DC | A(&SYSECT) |
| | | MEND | |
| | | MACRO | |
| | | OUTER1 | |
| 3 | CSOUT1 | CSECT | |
| | | DS | 100C |
| 4 | | INNER | INA |
| 5 | | INNER | INB |
| 6 | | DC | A(&SYSECT) |
| | | MEND | |
| | | MACRO | |
| | | OUTER2 | |
| 7 | | DC | A(&SYSECT) |
| | | MEND | |
| 8 | MAINPROG | CSECT | |
| | | DS | 200C |
| 9 | | OUTER1 | |
| 10 | | OUTER2 | |
| | MAINPROG | CSECT | |
| | | DS | 200C |
| | CSOUT1 | CSECT | |
| | | DS | 100C |
| | INA | CSECT | |
| | | DC | A(CSOUT1) |
| | INB | CSECT | |
| | | DC | A(INA) |
| | | DC | A(MAINPROG) |
| | | DC | A(INB) |

### 4.14.1.3 &SYSLIST

The system variable symbol &SYSLIST provides the programmer with an alternative to symbolic parameters for referring to positional macro instruction operands. It is particularly useful when operands are supplied in a positional macro instruction without a corresponding symbolic parameter appearing in the prototype of the macro definition.

&SYSLIST and symbolic parameters may be used in the same macro definition. &SYSLIST may also be used in place of symbolic parameters and follows all rules for concatenation with other characters that symbolic parameters obey. (See section 4.10, "Concatenation").

&SYSLIST(n) may be used to refer to the nth macro instruction operand. If the nth operand is a sublist, then &SYSLIST(n,m) may be used to refer to the mth operand in the sublist, where n and m may be any arithmetic expression allowed in the operand field of a SETA statement. This form of indexing may be extended to reference any element in an inner sublist in the same manner that was described for symbolic parameters.

If n equals zero, a null operand results. If n is greater than zero, the value of the macro instruction operand is given, provided that an operand exists corresponding to that value of n.

The attributes of any macro instruction operand or suboperand may be referenced, using the &SYSLIST notation, in the same manner as the attributes are referenced if a symbolic parameter is used. (Examples of this facility appear under the discussion of Attributes, section 4.15).



#### 4.14.1.4 &SYSVERS

The system symbol &SYSVERS represents the version number of the object program. It generates six bytes with the value 'VERxxx' (xxx = version number). x is a character value from 0 to 9. The value of &SYSVERS remains constant during assembly. This symbol is only meaningful if the source program to be assembled is stored in a source program library. If the source program is read from a file or is entered directly by way of SYSDTA, this system symbol cannot be evaluated. Its value in this case is as follows: VER␣␣␣.

In the following example the version number of the assembled object program is 14.

Using &SYSVERS

| Name | Operation | Operand | Comments |
|---|---|---|---|
| &NAME<br>&NAME | MACRO<br>VNUM<br>DC<br>MEND | C'&SYSVERS' | Macro definition |
| VERSION | VNUM | | Macro call |
| VERSION | DC | C'VER014' | Generated statement |

### 4.14.1.5 &SYSDATE

This system symbol represents the date of the translated source program. It generated 9 bytes with the value "mmddyynnn" (mm=month, dd=day, yy=year, nnn=day number of the year).

m, d, y, and n are character values between 0 and 9. The value of the symbol &SYSDATE remains constant during a translation.

In the following example, the date of the translated source program is 14 July 1981.

Application of &SYSDATE

| Name | Operation | Operands | Comments |
|---|---|---|---|
| &NAME<br>&NAME | MACRO<br>VDATE<br>DC<br>MEND | C'&SYSDATE' | Macro definition |
| DATE | VDATE | | Macro instruction |
| DATE | DC | C'071481195' | Generated Statement |

### 4.14.1.6 &SYSTIME

This system symbol represents the time of day of the assembled program. It generates 6 bytes with the value "hhmmss" (hh = hour, mm = minute, ss = second). h, m and s are character values between 0 and 9. The value of the time is computed at the beginning of the assembly and remains fixed during the assembly.

In the following example, the time of day at the start of assembly is 10h 34m 16s.

Application of &SYSTIME

| Name | Operation | Operands | Comments |
|---|---|---|---|
| &TIME<br>&TIME | MACRO<br>VTIME<br>DC<br>MEND | C'&SYSTIME' | Macro definition |
| TIME | VTIME | | Macro instruction |
| TIME | DC | C'103416' | Generated statement |

#### 4.14.1.7 &SYSVERM

The &SYSVERM system symbol represents the version number of the library macro in which it is used. The format is "VERxxx", where xxx = version number. In source program macros from MLU libraries, this value is "VER".

Application outside macros is not allowed.



#### 4.14.1.8 &SYSPARM

The &SYSPARM system symbol is an eight-byte CHARACTER variable. It is defined in a COMOPT statement and evaluated in the macro expansion.

Application of &SYSPARM: *COMOPT SYSPARM='ABCDEFG'

Application in the source program: &CG SETC '&SYSPARM'

In order to represent one apostrophe, two apostrophes must be written.

#### 4.14.1.9 &SYSTEM

The &SYSTEM system symbol represents the operating system version under which the assembly is executed.

It generates four bytes with the value "svvv", where

S = 1 = BS1000

S = 2 = BS2000

vvv = version no.

<u>Example:</u>

```
AIF ('&SYSTEM'(1,1) EQ '1').BS1
AIF ('&SYSTEM'(1,1) EQ '2').BS2
```

### 4.14.2 SET Symbols

The SET symbols are a class of variable symbols whose values may be set and changed by the programmer both inside and outside a macro definition. SET symbols must be defined by the programmer before they are used. When a SET symbol is defined, it is assigned an initial value. SET symbols may be assigned new values by means of the SETA, SETB, and SETC instructions (see also Default Values, Appendix A.9 item 10). A SET symbol is defined when it appears as an operand of an LCLA, LCLB, LCLC, GBLA, GBLB, or GBLC instruction. A SET symbol is classified as local or global, according to the type of statement that is used to define it.

#### 4.14.2.1 Defining SET Symbols

SET symbols must be defined by the programmer before they are used in conditional and SET instructions inside or outside macro definitions.

The LCLA, LCLB, and LCLC instructions are used to define and set initial values to SETA, SETB, and SETC symbols, respectively, which are to be used as local SET symbols.

The GBLA, GBLB and GBLC instructions are used to define and set initial values to SETA, SETB, and SETC symbols, respectively, which are to be used as global SET symbols.

**1. LCLA, LCLB, LCLC - Define Local SET Symbols**

The format of these instructions is:

| Name | Operation | Operand |
|---|---|---|
| Blank | LCLA, LCLB or LCLC | One or more variable symbols that are to be used as local SET symbols, separated by commas. |

The SETA, SETB, or SETC symbols written in the operand entry are assigned the initial values 0.

If an LCLA, LCLB, or LCLC instruction appears outside macro definitions, then the values of the variable symbols written in the operand entry will be local to the source program and cannot be referenced in a macro definition.

If an LCLA, LCLB, or LCLC instruction appears within a macro definition, then the values of the variable symbols written in the operand entry will be local to that macro definition and cannot be referenced in the source program or in any other macro definition.

If the same variable symbol is defined as local in the source program and in a macro definition, or in two or more macro definitions, the symbol is considered to be a different SET symbol in each case.

A variable symbol may not be used as a symbolic parameter and as a local SET symbol in the same definition. No SET symbols must be used that begin with the character sequence &SYS (see 4.14.1).

All LCLA, LCLB, or LCLC instructions in a macro definition must appear immediately after the prototype statement and after all GBLA, GBLB, or GBLC instructions. All LCLA, LCLB, or LCLC instructions outside macro definitions in the source program must appear after all GBLA, GBLB, and GBLC instructions outside macro definitions, before all conditional assembly instructions, before all SET instructions, and before the first control section of the program.

**2. GBLA, GBLB, GBLC - Define Global SET Symbols**



The format of these instructions is:

| Name | Operation | Operand |
|---|---|---|
| Blank | GBLA, GBLB, or GBLC | One or more variable symbols that are to be used as global SET symbols, separated by commas. |

The GBLA, GBLB, and GBLC instructions define global SETA, SETB, and SETC symbols, respectively, and assign the same initial values as the corresponding types of local SET symbols. However, a global SET symbol is assigned an initial value by only the first GBLA, GBLB, or GBLC instruction processed in which the symbol appears.
Subsequent GBLA, GBLG, or GBLC instructions processed by the assembler do not affect the value assigned to the SET symbol. No SET symbols must be used that begin with the character sequence &SYS (see 4.14.1).

GBLA, GBLB or GBLC instructions may appear both inside and outside macro definitions.

All GBLA, GBLB, and GBLC instructions in a macro definition must appear before all LCLA, LCLB, and LCLC instructions in that macro definition.
All GBLA, GBLB, and GBLC instructions outside macro definitions must appear before all LCLA, LCLB, and LCLC instructions outside macro definitions.

### 4.14.2.2 Using SET Symbols

When a SET symbol appears in an assembly language statement inside or outside macro definitions, the current value of the SET symbol replaces the SET symbol in the statement. The resulting statement must be a valid assembly language statement. Variable symbols may not be used to generate other variable symbols, nor may SET symbols appear in a macro definition prototype statement. Variable symbols may be used in the operation code entry of a statement under the restrictions previously stated (see the discussion of "Model Statements", section 4.8).

The same variable symbol may not be used as a symbolic parameter and as a SET symbol in the same macro definition.

The following example illustrates this rule.

| Name | Operation | Operand |
|---|---|---|
| &NAME | MOVE | &TO,&FROM |

If the statement above is a prototype statement, then &NAME, &TO, and &FROM may not be used as SET symbols in the macro definition.

The same variable symbol may not be used as two different types of SET symbols in the same macro definition. Similarly, the same variable symbol may not be used as two different types of SET symbols outside macro definitions.

For example, if &A is a SETA symbol in macro definition, it cannot be used as a SETC symbol in that definition. Similarly, if &A is a SETA symbol outside macro definitions, it cannot be used as a SETC symbol outside macro definitions.

The same variable symbol if declared local may be used in two or more macro definitions and outside macro definitions. If this is the case, the variable symbol will be considered a different variable symbol each time it is used.

For example, if &A is a variable symbol (either SET symbol or symbolic parameter) in one macro definition it can be used as a variable symbol (either SET symbol or symbolic parameter) in another definition. Similarly, if &A is a variable symbol (SET symbol or symbolic parameter) in a macro definition, it can be used as a SET symbol outside macro definitions.

If a local SET symbol is defined in two or more macro definitions, or in a macro definition and outside macro definitions, the SET symbol is considered to be a different SET symbol in each case. A global SET symbol, however, is the same SET symbol in each place it is defined.

Global SET symbols are the only global variable symbols. Global SET symbols may communicate values between one or more macro definitions, and between macro definitions and the source program. Local SET symbols, however, communicate values between statements in the same macro definition, or between statements outside macro definitions in the source program.

A SET symbol must be defined as a global SET symbol in each macro definition in which it is to be used as a global SET symbol. If it is also to be used as a global SET symbol outside macro definitions, it must be defined as a global SET symbol outside macro definition as well.

If the same SET symbol is defined as a global SET symbol in one or more places, and as a local SET symbol elsewhere, it is considered the same symbol wherever it is defined as a global SET symbol, and a different symbol in each place it is defined as a local SET symbol.

All variable SET symbols and system variable symbols may be concatenated with other characters. It should be noted that a period is not required to concatenate a SET symbol or system variable symbol with a left parenthesis, unless the SET symbol is defined as subscripted (see Subscripted Set Symbols, section 4.17.5).

Variable symbols in macro instructions are replaced by the values assigned to them immediately prior to the start of processing the macro definition. If a SET symbol appears in the operand entry of a macro instruction, attribute information will not be provided.

The (SET(A), SET(B), and SET(C) instructions may be used to change the values of SETA, SETB, and SETC symbols, respectively. A SET(C) statement may also be used to change the value of a symbolic parameter. The type designation, written in parentheses above, is optional. The SET instructions are described following the description of attribute references below. Attribute references may appear in all SET and conditional statements.

## 4.15 ATTRIBUTES

The assembler assigns attributes to macro instruction operands and to symbols that name statements in the source program. Attribute references may appear only in conditional assembly and SET instructions; i.e., the AIF and SET instructions.

There are six kinds of attributes. They are: type, length, scaling, integer, count and number. The count and number attributes may only be used within a macro definition.

Each attribute has a notation associated with it. The notations are:

| **Attribute:** | **Notation:** |
|---|---|
| Type | T' |
| Length | L' |
| Scaling | S' |
| Integer | I' |
| Count | K' |
| Number | N' |

In a conditional statement that is outside macro definitions, an attribute is referenced by writing the appropriate attribute notation immediately followed by a symbol (e.g., T'NAME refers to the type attribute of the symbol NAME).

In a conditional statement that is within a macro definition, an attribute is referenced by writing the appropriate attribute notation immediately followed by a symbolic parameter, (e.g., L'&PARAM refers to the length attribute of the macro instruction operand that corresponds to the symbolic parameter &PARAM). If the operand is a sublist, L'&PARAM(2) may be used to refer to the length attribute of the second element of the sublist.

Attributes are assigned to symbols according to the following rules:

1. The symbol must appear in the name entry of an assembly language statement or in the operand of an EXTRN statement in the source program. The statement must appear outside macro definitions.

2. If variable symbols appear within a statement, in a field which must be processed to calculate an attribute, the attribute will be undefined, unless the statement is generated prior to the attribute reference.

3. Attributes will be undefined for DC or DS statements which contain expressions in modifier fields.

   Example:

   (A+B)XL;'01'CL(5*C)'AC'

4. The attributes of symbols are undefined in the following cases:

   a) if the symbol names a macro-generated statement,

   b) if the symbol appears in the name entry of two or more statements in the source program, even if only one statement will appear in the assembled program,

   c) if the symbol appears as an operand or suboperand of an inner macro instruction,

   d) if the symbol is generated by a variable symbol,

   e) if the symbol names an EQU statement without specification of length or type (see 3.26.2) or an LTROG statement,

   f) if the symbol is a standard value supplied in a keyword macro prototype.

If an inner macro instruction operand is a symbolic parameter, then attributes of the operand are the same as the attributes of the corresponding outer macro instruction operand.

If a macro instruction operand is an expression beginning with a valid symbol, the operand gets the attributes of this symbol.

If a macro instruction operand is a sublist, the programmer may refer to the attributes of either the sublist or each operand in the sublist. The type, length, scaling and integer attributes of a sublist are the same as the corresponding attributes of the first element of the sublist. The number and count attributes of a sublist are discussed below.

Each category of attribute will now be discussed in detail.

### 4.15.1 Type Attribute (T')

The type attribute of a macro instruction operand or a symbol is a letter.

The programmer may refer to a type attribute in the operand of SETC instructions or in character relations in the operands of SETB, or AIF(B) instructions.

The following letters are used for symbols that name DC, DS, or DXD statements and for outer macro instruction operands that are symbols that name DC, DS, or DXD statements.

A A-type address constant, implied length, aligned to fullword boundary, as well as CXD.

B Binary constant.

C Character constant.

D Long floating-point constant, implied length, aligned to doubleword boundary.

E Short floating-point constant, implied length, aligned.

F Full-word fixed-point constant, implied length, aligned to fullword boundary.

G Fixed-point constant, explicit length.

H Half-word fixed-point constant, implied length, aligned to halfword boundary.

K Floating-point constant, explicit length.

L Floating-point constant, extended precision, implied length, aligned.

P Packed decimal constant.

Q Relocatable address in an external dummy section.

R A-, S-, V-, or Y-type address constant, explicit length.

S S-type address constant, implied length, aligned.

V V-type address constant, implied length, aligned.

X Hexadecimal constant.

Y Y-type address constant, implied length, aligned to halfword boundary.

Z Zoned decimal constant.

The following letters are used for symbols (and outer macro instruction operands that are symbols) that name statements other than DC or DS statements, or that appear in the operand field of an EXTRN statement.

I Machine instruction.

J Control section name.

M Macro instruction.

T External symbol.

The following letters are used for inner and outer macro instruction operands only.

N Self-defining term, SETA and SETB variable

O Omitted operand.

The letter U (undefined) is used for inner and outer macro instruction operands that cannot be assigned any of the above letters, such as literals or character strings, for all system symbols and SETC variables. The letter U is also assigned to ordinary symbols in the case outlined in Attribute Rule 4 above.

The type attribute of macro instruction operands that are generated by symbolic parameters other than variable symbols is U.

The type attributes of all operands in the following examples are undefined:

| | Name | Operation | Operand |
|---|---|---|---|
| Outer macro Instruction | | CALL1 | 'BOY',E(F=G),ABCDEFGHI,=A(O) |
| Inner Macro Instruction | | CALL2 | CAT |

The attributes in the following examples will be defined only if the statements are defined prior to a reference to the attribute.

A DC &X.L2'1' where &X is a character variable symbol.

B DC &B.F'01' where &B is an arithmetic variable symbol.

### 4.15.2 Length (L'), Scaling (S'), and Integer (I') Attributes

The length, scaling, and integer attributes of macro instruction operands and symbols are numeric values. The length attribute of a symbol (or a macro instruction operand that is a symbol) is as described in section 3.8 of this publication. The length attribute of a symbol or macro instruction operand whose type attribute is M, O or U is zero. If the type attribute is N or T, the length attribute is one.

Exception:

Any length may be specified for EQU statements in extended format (see 3.26.2).

Scaling and integer attributes are provided for symbols that name fixed-point, floating-point, and decimal DC or DS statements.

**Fixed and Floating Point:** The scaling attribute of a fixed point or floating point number is the value given by the scale modifier. The integer attribute is a function of the scale and length attributes of the number.

**Decimal:** The scaling attribute of a decimal number is the number of decimal digits to the right of the decimal point. The integer attribute of a decimal number is the number of decimal digits to the left of the decimal point.

Scaling and integer attributes are available for symbols and macro instruction operands only if their type attributes are H, F, and G (fixed-point); D, E, and K (floating-point); or P and Z (decimal).

The programmer may refer to the length, scaling, and integer attributes in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of SETB or AIF instructions.

### 4.15.3 Count Attribute (K')

The programmer may refer to the count attribute of macro instruction operands and to the count attribute of SETA, SETB and SETC variables and system symbols.

The count attribute of macro instruction operands is a value equal to the number of characters in the macro instruction operand after substitution for variable symbols. If the operand is a sublist, the count attribute includes the beginning and ending parentheses and the commas within the sublist. The count attribute of any suboperand of any level sublist may be referenced. The count attribute of an omitted operand is zero.

If a macro instruction operand contains variable symbols, the characters that replace the variable symbols, rather than the variable symbols, are used to determine the count attribute.

**The count attribute for system symbols and SET variables:**

| | |
|---|---|
| &SYSDATE | 9 |
| &SYSTIME | 6 |
| &SYSTEM | 4 |
| &SYSVERS | 6 |
| &SYSVERM | 6 |
| &SYSNDX | 4 |
| &SYSPARM | Number of characters from corresponding COMOPT option (up to 8) |
| &SYSECT | Number of characters that the name in the variable &SYSECT has (up to 8) |
| SETA variable | Number of characters required to indicate the current number value of the SET symbol as a decimal number, without leading zeros. |
| | E.g. &AG1 SETA 111 K'AG1=3<br>&AG2 SETA X'FF' K'AG2=3 |
| SETB variable | Constant value 1 |
| SETC variable | Number of character containing the variable |

The programmer may refer to the count attribute in the operand field of a SETA instruction , ot in arithmetic relations in the operand fields of SETB and AIF instructions.

### 4.15.4 Number Attribute (N')

The programmer may refer to the number attribute of macro instruction operands only.

The number attribute is a value equal to the number of operands in the operand list (&SYSLIST) or in an operand sublist. The number of operands in an operand sublist is equal to one plus the number of commas that indicate the end of an operand in the sublist. The number attribute of any nested sublist may be referenced.

If the macro instruction operand is not &SYSLIST and not a sublist, the number attribute is one. If the macro instruction operand is omitted, the number attribute is zero.

The following examples illustrate the N' and K' attributes for various macro instruction operands within the sublist corresponding to the symbolic parameter &P.

| Operand | Value: | N' | K' |
|---|---|---|---|
| &P | (A,(A,B),(C,(D,E)),G) | 5 | 22 |
| &P(1) | A | 1 | 1 |
| &P(2) | (A,B) | 2 | 5 |
| &P(3) | (C,(D,E)) | 2 | 9 |
| &P(4) | G | 1 | 1 |
| &P(5) | NULL | 0 | 0 |
| &P(2,2) | B | 1 | 1 |
| &P(3,2) | (D,E) | 2 | 5 |
| &P(3,2,1) | D | 1 | 1 |
| &P(1,1) | A | 1 | 1 |
| &P(2,3) | NULL | 0 | 0 |
| &SYSLIST | A,B,C | 3 | N/A |

The programmer may refer to the number attribute in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of SETB and AIF instructions that are part of a macro definition.

## 4.16 ASSIGNING INTEGER ATTRIBUTES TO SYMBOLS

The integer attribute is computed from the length and scaling attributes.

**Fixed-Point:** The integer attribute of a fixed-point number is equal to eight times the length attribute of the number minus the scaling attribute minus one; i.e., I'=8*L'-S'-1.

Each of the following statements defines a fixed-point field. The length attribute of HALFCON is 2, the scaling attribute is 6, and the integer attribute is 9. The length attribute of ONECON is 4, the scaling attribute is 8, and the integer attribute is 23.

| Name | Operation | Operand |
|---|---|---|
| HALFCON | DC | HS6'-25.93' |
| ONECON | DC | FS8'100.3E-2' |

**Floating Point:** The integer attribute of a floating-point number is equal to two times the difference between the length attribute of the number and one, minus the scaling attribute; i.e., I'=2*(L'-1)-S'.

EAch of the following statements defines a floating-point value. The length attribute of SHORT is 4, the scaling attribute is 2, and the integer attribute is 4. The length attribute of LONG is 8, the scaling attribute is 5, and the integer attribute is 9.

| Name | Operation | Operand |
|---|---|---|
| SHORT | DC | ES2'46.415' |
| LONG | DC | DS5'-3.729' |

**Decimal:** The integer attribute of a packed decimal number is equal to two times the length attribute of the number minus the scaling attribute minus one; i.e. I'=2*L'-S'-1. The integer attribute of a zoned decimal number is equal to the difference between the length attribute and the scaling attribute; i.e., I'=L'-S'.

Each of the following statements defines a decimal field. The length attribute of FIRST is 2, the scaling attribute is 2, and the integer attribute is 1. The length attribute of SECOND is 3, the scaling attribute is 0, and the integer attribute is 3. The length attribute of THIRD is 4, the scaling attribute is 2, and the integer attribute is 2. The length attribute of FOURTH is 3, the scaling attribute is 2, and the integer attribute is 3.

| Name | Operation | Operand |
|---|---|---|
| FIRST | DC | P'+1.25' |
| SECOND | DC | Z'-543' |
| THIRD | DC | Z'79.68' |
| FOURTH | DC | P'79.68' |

## 4.17 SET INSTRUCTIONS

### 4.17.1 SETA - Set Arithmetic

The SETA (or SET) instruction may be used to assign an arithmetic value to a SETA symbol. The value of the SETA variable symbol may be changed by a subsequent SETA statement.

The format of this instruction is:

| Name | Operation | Operand |
|---|---|---|
| A SETA symbol | SETA, or SET | An arithmetic expression |

The expression in the operand entry is evaluated as a signed 32-bit arithmetic value which is assigned to the SETA symbol in the name entry. The minimum and maximum allowable values of the expression are $-2^{31}$ and $+2^{31}-1$, respectively.

The expression may consist of one term or an arithmetic combination of terms. The terms that may be used are self-defining terms, variable symbols, character values, and the length, scaling, integer, count and number attributes.

Furthermore, the value may be an empty string, e.g. the value of an unoccupied macro operand. An empty string is interpreted as zero.

Notes:

- A SETC variable symbol or character value may be used as a term in a SETA expression if the value is from 1 to 8 decimal digits. A symbolic parameter may be used if its value is a decimal, hexadecimal or binary self-defining term.

- The arithmetic operators that may be used to combine the terms of an expression are + (addition), - (subtraction), * (multiplication), and / (division).

- An expression may not contain two terms or two dyadic operators in succession. The monadic operators + and - may appear before any term and at the beginning of an expression.

The following are valid operand fields at SETA instructions:

```
&AREA+X'2D'    I'&N/25
&BETA*10       &EXIT-S'&ENTRY+1
L'&HERE+32     29
&Q+'10'
+(-20+-10)
```

The following are invalid operand fields of SETA instructions:

```
&AREAX'C'      (two terms in succession)
&FIELD+*       (two operators in succession)
-&DELTA*2      (begins with an operator)
*+32           (begins with an operator; two operators in succession)
NAME/15        (NAME is not a valid term)
```



### 4.17.1.1 Evaluation of Arithmetic Expressions

The procedure used to evaluate the arithmetic expression in the operand of a SETA instruction is the same as that used to evaluate arithmetic expressions in assembly language statements. The only difference between the two types of arithmetic expressions is the terms that are allowed in each expression.

The following evaluation procedure is used:

1. Each term is given its numerical value.

2. The arithmetic operations are performed moving from left to right. However, multiplication and/or division are performed before addition and subtraction.

3. The computed result is the value assigned to the SETA symbol in the name entry.

The arithmetic expression in the operand entry of a SETA instruction may contain one or more sequences of arithmetically combined terms that are enclosed in parentheses. A sequence of parenthesized terms may appear within another parenthesized sequence. The following are examples of SETA instruction operands that contain parenthesized sequences of terms.

```
(L'&HERE+32)*29
&AREA+X'2D'/(&EXIT-S'&ENTRY+1)
&BETA*10*(I'&N/25/(&EXIT-S'&ENTRY+1))
```

The parenthesized portion or portions of an arithmetic expression are evaluated before the rest of the terms in the expression are evaluated. If a sequence of parenthesized terms appears within another parenthesized sequence, the innermost sequence is evaluated first.

As the SETA expression is scanned, a counter is incremented by one for each term that is recognized. When an arithmetic operation is possible, according to the operator precedence rules, it is performed, and the counter is decremented by one. The process continues until the expression has been completely evaluated or until the counter exceeds four.

### 4.17.1.2 Using Local SETA Symbols

The arithmetic value assigned to a SETA symbol is substituted for the SETA symbol when it is used in an arithmetic relation. If the SETA symbol is not used in an arithmetic expression, the arithmetic value is completely converted to an unsigned integer, with leading zeros removed. If the value is zero, it is converted to a single zero.

The following example illustrates this rule:

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &NAME | MOVE | &TO,&FROM |
| | | LCLA | &A,&B,&C,&D |
| 1 | &A | SETA | 10 |
| 2 | &B | SETA | 12 |
| 3 | &C | SETA | &A-&B |
| 4 | &D | SETA | &A+&C |
| | &NAME | ST | 2,SAVEAREA |
| 5 | | L | 2,&FROM&C |
| 6 | | ST | 2,&TO&D |
| | | L | 2,SAVEAREA |
| | | MEND | |
| | HERE | MOVE | FIELDA,FIELDB |
| | HERE | ST | 2,SAVEAREA |
| | | L | 2,FIELDB2 |
| | | ST | 2,FIELDA8 |
| | | L | 2,SAVEAREA |

Statements 1 and 2 assign to the SETA symbols &A and &B the arithmetic values +10 and +12, respectively. Therefore, statement 3 assigns the SETA symbol &C the arithmetic value -2. When &C is used in statement 5, the arithmetic value -2 is converted to the unsigned integer 2. When &C is used in statement 4, however, the arithmetic value -2 is used. Therefore, &D is assigned the arithmetic value +8. When &D is used in statement 6, the arithmetic value +8 is converted to the unsigned integer 8.

The following example shows how the value assigned to a SETA symbol may be changed in a macro definition.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &NAME | MOVE | &TO,&FROM |
| | | LCLA | &A |
| 1 | &A | SETA | 5 |
| | &NAME | ST | 2,SAVEAREA |
| 2 | | L | 2,&FROM&A |
| 3 | &A | SETA | 8 |
| 4 | | ST | 2,&TO&A |
| | | L | 2,SAVEAREA |
| | | MEND | |

| | Name | Operation | Operand |
|---|---|---|---|
| | HERE | MOVE | FIELDA,FIELDB |
| | HERE | ST | 2,SAVEAREA |
| | | L | 2,FIELDB5 |
| | | ST | 2,FIELDA8 |
| | | L | 2,SAVEAREA |

Statement 1 assigns the arithmetic value +5 to SETA symbol &A. In statement 2, &A is converted to the unsigned integer 5. Statement 3 assigns the arithmetic value +8 to &A. In statement 4, therefore, &A is converted to the unsigned integer 8, instead of 5.

A SETA symbol or SETA arithmetic expression may be used with a symbolic parameter to refer to an operand in a sublist.

The following macro definition may be used to add the last operand in an operand sublist to the first operand in a sublist and store the result at the first operand. A sample macro instruction and generated statements follow the macro definition.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| 1 | | ADDX | &NUMBER,&REG |
| | | LCLA | &LAST |
| 2 | &LAST | SETA | N'&NUMBER |
| | | L | &REG,&NUMBER(1) |
| 3 | | A | &REG,&NUMBER(&LAST) |
| | | ST | &REG,&NUMBER(1) |
| | | MEND | |
| 4 | | ADDX | (A,B,C,D,E),3 |
| | | L | 3,A |
| | | A | 3,E |
| | | ST | 3,A |

&NUMBER is the first symbolic parameter in the operand entry of the prototype statement (statement 1). The corresponding characters, (A,B,C,D,E), of the macro instruction (statement 4) are a sublist. Statement 2 assigns to &LAST the arithmetic value +5, which is equal to the number of operands in the sublist.

Therefore, in statement 3, &NUMBER(&LAST) is replaced by the fifth operand of the sublist.

A SETA symbol or SETA arithmetic expression may be used with &SYSLIST to refer to positional macro operands, as illustrated after the discussion of the MEXIT instruction.

SETA symbols used as global variable symbols are illustrated in a subsequent section entitled "Using Global SET Symbols" (see section 4.17.4).

### 4.17.2 SETC - SET Character

The SETC (or SET) instruction is used to assign a character value to a SETC symbol, to change a previously assigned value, or to change the the value of a symbolic parameter.

The format of this instruction is:

| Name | Operation | Operand |
|---|---|---|
| A SETC symbol or symbolic parameter | SETC | One operand, of the form described below |

The operand may consist of the type attribute, a character value, a substring notation, or a concatenation of substring notations and character expressions. Variable symbols may appear in the operand field of a SETC statement. If a SETA or SETB symbol is used in a SETC statement, the result is the character representation of the decimal or boolean value, respectively, unsigned, with leading zeros removed. If the value is zero, one decimal zero is used.

The terms allowed in a SETC instruction are described below.

#### 4.17.2.1 Type Attribute

The character value assigned to a SETC symbol or symbolic parameter may be a type attribute. If the type attribute is used, it must appear alone in the operand field. The following example assigns to the SETC symbol &TYPE the letter that is the type attribute of the macro instruction operand that corresponds to the symbolic parameter &ABC.

| Name | Operation | Operand |
|---|---|---|
| &TYPE | SETC | T'&ABC |

#### 4.17.2.2 Character Value

A character value or expression consists of any combination of characters enclosed in apostrophes and preceded by a duplication factor, if required. The duplication factor is an arithmetic expression that may be used as an operand in a SETA instruction. It is enclosed in parentheses. A SETC symbol or symbolic parameter can be assigned a character value of not more than 127 characters after execution of the duplication factor. Intermediate values of character expressions from which the final value is extracted may be up to 255 characters. Concatenation also permits up to 255 characters to be assigned.

Example of a character expression with duplication factor:

(3)'A' produces 'AAA'

### 4.17.2.3 Evaluation of Character Expressions

The following statement assigns the character value AB%4 to the SETC symbol &ALPHA:

| Name | Operation | Operand |
|---|---|---|
| &ALPHA | SETC | 'AB%4' |

More than one character expression may be concatenated into a single character expression by placing a period between the terminating apostrophe of one character expression and the opening apostrophe of the next character expression. For example, either of the following statements may be used to assign the character value ABCDEF to the SETC symbol &BETA.

| Name | Operation | Operand |
|---|---|---|
| &BETA | SETC | 'ABCDEF' |
| &BETA | SETC | 'ABC'.'DEF' |

Two apostrophes must be used to represent an apostrophe that is part of a character expression.

The following statement assigns the character value L'SYMBOL to the SETC symbol &LENGTH.

| Name | Operation | Operand |
|---|---|---|
| &LENGTH | SETC | 'L"SYMBOL' |

Variable symbols may be concatenated with other characters in the operand field of a SETC instruction according to the general rules for concatenating variable symbols with other characters.

If &ALPHA has been assigned the character value AB%4, the following statements may be used to assign the character value AB%4RST to the variable symbol &GAMMA.

| Name | Operation | Operand |
|---|---|---|
| &GAMMA | SETC | '&ALPHA.RST' |

or

| Name | Operation | Operand |
|---|---|---|
| &GAMMA | SETC | '&ALPHA'.'RST' |

Two ampersands must be used to represent an ampersand that is not part of a variable symbol. Both ampersands become part of the character value assigned to the SETC symbol. They are not replaced by a single ampersand.

The following statement assigns the character value HALF&& to the SETC symbol &AND.

| Name | Operation | Operand |
|---|---|---|
| &AND | SETC | 'HALF&&' |

In this example,

| Name | Operation | Operand |
|---|---|---|
| &A | SETC | '&&BETA'(2,5) |

'&&BETA'(2,5) produces &BETA which is considered a character string, not a variable symbol.



#### 4.17.2.4 Substring Notation

The value assigned to a SETC symbol or symbolic parameter may be a substring. Substring character values permit the programmer to assigns part of a character value to a variable symbol.

If the programmer wants to assign part of a character value to a SETC symbol, he must indicate to the assembler in the operand of a SETC instruction:

1. the character value itself, and

2. the part of the character value he wants to assign to the SETC symbol or symbolic parameter.

The concatenation of 1. and 2. in the operand of a SETC instruction is called a substring notation. The character value that is assigned to the variable symbol name entry is called a substring character value.

Substring notation consists of a character expression, immediately followed by two arithmetic expressions that are separated from each other by a comma and are enclosed in parentheses. The two arithmetic expressions may be any expression that is allowed in the operand of a SETA instruction, with the restriction that character values are not permitted.

The first expression indicates the first character (in the character expression) that is to be assigned to the SETC symbol or symbolic parameter in the name entry of the SETC statement. The second expression indicates the number of consecutive characters in the character expression (starting with the character indicated by the first expression) that are to be assigned to the SETC symbol. If a substring specifies more characters than are in the character string, the number of available characters will be supplied.

The maximum size character expression from which the substring character value can be chosen, is 255 characters.

A duplication factor can be applied to a substring. In a substring notation, first the substring is evaluated and then the duplication factor is executed.

Example:

(2)'ABC'(3,1) produces 'CC'

The following are valid substring notations:

```
'&ALPHA'(2,5)
'AB%4'(&AREA+2,1)
'&ALPHA'.'RST'(6,&A)
'ABC&GAMMA'(&A,&AREA+2)
(&A)'&ALPHA'(&B,&C) where &A,&B,&C are arithmetic expressions.
```

The following are invalid substring notations:

| | |
|---|---|
| '&BETA' (4,6) | (blanks between character value and arithmetic expressions) |
| 'L''SYMBOL'(142-&XYZ) | (only one arithmetic expression) |
| 'AB%4&ALPHA'(&&FIELD*2) | (arithmetic expression not separated by a comma) |
| 'BETA'4,6 | (arithmetic expressions not enclosed in parentheses) |
| '&ALPHA'(2,4)(1,1) | (double substring notation is not permitted) |

### 4.17.2.5 Concatenating Substring Notations and Character Expressions

Substring notations may be concatenated with character expressions in the operand of a SETC instruction. If a substring notation follows a character expression, the two may be concatenated by placing a period between the terminating apostrophe of the character expression and the opening apostrophe of the substring notation.

For example, if &ALPHA has been assigned the character value AB%4, and &BETA has been assigned the character value ABCDEF, then the following statement assigns &GAMMA the character value AB%4BCD.



| Name | Operation | Operand |
|---|---|---|
| &GAMMA | SETC | '&ALPHA'.'&BETA'(2,3) |

If a substring notation precedes a character expression or another substring notation, the two may be concatenated by writing a period immediately after the closing parenthesis of the substring notation. The period may, however, be omitted if the character expression following the substring notation does not begin with a duplication factor.

If &ALPHA has been assigned the character value AB%4, and &ABC has been assigned the character value 5RS, either of the following statements may be used to assign &WORD the character value AB%45RS.

| Name | Operation | Operand |
|---|---|---|
| &WORD | SETC | '&ALPHA'(1,4)'&ABC' |
| &WORD | SETC | '&ALPHA'(1,4)'&ABC'(1,3) |

If a SETC symbol is used in the operand of a SETA instruction, the character value assigned to the SETC symbol must be one to eight decimal digits.

If a SETA symbol is used in the operand of a SETC statement, the arithmetic value is converted to a single zero. If a SETB symbol is used in the binary operand of a SETC statement, the value is converted to the character representation of one or zero.

### 4.17.2.6 Using Local SETC Symbols

The character value assigned to a SETC symbol or symbolic parameter is substituted for that variable symbol when it is used in the name, operation, or operand of a statement.

For example, consider the following macro definition, macro instruction, and generated statements.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &NAME | MOVE | &TO,&FROM |
| | | LCLC | &PREFIX |
| 1 | &PREFIX | SETC | '&TO'(1,4) |
| | &NAME | ST | 2,SAVEAREA |
| 2 | | L | 2,&PREFIX&FROM |
| | | ST | 2,&TO |
| | | L | 2,SAVEAREA |
| | | MEND | |
| | HERE | MOVE | FIELDA,B |
| | HERE | ST | 2,SAVEAREA |
| | | L | 2,FIELDB |
| | | ST | 2,FIELDA |
| | | L | 2,SAVEAREA |

Statement 1 assigns the substring character value FIELD (the first five characters corresponding to symbolic parameter &TO) to the SETC symbol &PREFIX . Therefore, FIELD replaces &PREFIX in statement 2.

Examples of the use of the SETC instruction to change the value of a symbolic parameter appear in a later section of this manual.

### 4.17.3 SETB - SET Binary

The SETB instruction may be used to assign the binary value 0 or 1 to a SETB symbol.

The format of this instruction is:

| Name | Operation | Operand |
|---|---|---|
| A SETB symbol | SETB or SET | A 0 or a 1, (0) or (1), or a logical expression enclosed in parentheses |

The operand may contain a 0 or a 1 or a logical expression enclosed in parentheses. A logical expression is evaluated to determine if it is true or false; the SETB symbol in the name entry is then assigned the binary value 1 or 0 corresponding to true or false, respectively.

A logical expression consists of one term or a logical combination of terms. The terms that may be used alone or in combination with each other are arithmetic relations, character relations, and SETB symbols.

The logical operators used to combine the terms of an expression are AND, OR, and NOT. The NOT operator must be preceded by AND or OR, unless it is the first operator of a logical expression.

A logical expression may not contain two terms in succession. A logical expression may contain two operators in succession only if the first operator is either AND or OR, and the second operator is NOT. A logical expression may begin with the operator NOT. It may not begin with the operators AND or OR.



An arithmetic relation consists of two arithmetic expressions, or an arithmetic and a character expression connected by a relational operator. The relational operators are EQ (equal), NE (not equal), LT (less than), GT (greater than, LE (less than or equal), and GE (greater than or equal). Any expression that is permissible in the operand of a SETA instruction, may be used as an arithmetic expression in the SETB operand entry, including SETA symbols, symbolic parameters, and character values, under the restrictions previously discussed.

A character relation consists of two character values connected by a relational operator. Anything that may be used in the operand of a SETC instruction may be used as a character value in the operand of a SETB instruction. It is necessary to enclose SETC symbols, &SYSECT, &SYSLIST, or symbolic parameters in quotes when they are used in character relations.

The type of expressions and terms that are used in a relation determines the nature of the comparison used to evaluate it. A logical relation results when all terms are considered to be character; that is, all terms are valid operands of SETC instructions, SETC symbols, the system variable symbols &SYSECT or &SYSLIST, or symbolic parameters. In all other cases, an arithmetic relation results.

The following examples illustrate these rules. &CG4 and &CG2 are character variable symbols. &AL4, &AL10, &AG1 and &AG3 are arithmetic variable symbols. &AREA is a symbolic parameter.

| | Type of Relation |
|---|---|
| ('FIELD' NE '&CG4') | Character |
| ('FIELD' NE &CG4) | Character |
| (&AL4 GT &AG1) | Arithmetic |
| (&AL4 GT '&AG1') | Arithmetic |
| (&CG4 LE &CG2) | Character |
| ('&CG2.X9' EQ 'BOY') | Character |
| (&AL10+&AL4*7 LT 16*&AG3+4) | Arithmetic |
| (&AL10 EQ '&CG2'(3,&AG1)) | Arithmetic |
| (&AREA+2 GT 29) | Arithmetic |
| (&AREA EQ &CG4) | Character |

The maximum size character values that can be compared is 127 characters. If the two character values are of unequal length, then the shorter one will always compare less than the longer one, regardless of the characters present.

The relational and logical operators must be immediately preceded and followed by at least one blank or other special character. Each relation may or may not be enclosed in parentheses. If a relation is not enclosed in parentheses, it must be separated from the logical operators by at least one blank or other special character.

A relation enclosed in parentheses need not be separated from the parentheses by any blanks.

The following are valid operand fields of SETB instructions:

```
(&AREA+2 GT 29)
('AB%4' EQ '&ALPHA')
(T'&ABC NE T'&XYZ)
(T'&P12 EQ 'F')
(&AREA+2 GT 29 OR &B)
(NOT &B AND &AREA+X'2D' GT 29)
('&C 'EQ' MB')
```

The following are invalid operand fields of SETB instructions:

```
&B                          (not enclosed in parentheses)
(T'&P12 EQ 'F' &B)          (two terms in succession)
('AB%4' EQ 'ALPHA' NOT &B)  (the NOT operator must be preceded by AND or OR)
(AND T'&P12 EQ 'F')         (expression begins with AND)
```

### 4.17.3.1 Evaluation of Logical Expressions

The following procedure is used to evaluate a logical expression in the operand field of a SETB instruction:

1. Each term (i.e., arithmetic relation, character relation, or SETB symbol) is evaluated and given its logical value (true or false). If an arithmetic term is combined in a relation, the character term will be converted to its arithmetic value before the relation is evaluated. If the character term does not contain only decimal characters, its arithmetic value is zero.

2. The logical operations are performed moving from left to right. However, NOTs are performed before ANDs, and ANDs are performed before ORs.

3. The computed result is the value assigned to the SETB symbol in the name field.

The logical expression in the operand of a SETB instruction may contain one or more sequences of logically combined terms that are enclosed in parentheses. A sequence of parenthesized terms may appear within another parenthesized sequence.

The following are examples of SETB instruction operands that contain parenthesized sequences of terms.

```
(NOT(&B AND &AREA+X'2D' GT 29)
(&B AND (T'P12 EQ 'F') OR &B)
```

The parenthesized portion or portions of a logical expression are evaluated before the rest of the terms in the expression are evaluated. If a sequence of parenthesized terms appears within another parenthesized sequence, the innermost sequence is evaluated first.

A maximum of 8 levels of parentheses are permitted.

Logical expressions follow the rule that was given for arithmetic expressions for calculating the maximum number of permissible terms.

### 4.17.3.2 Using Local SETB Symbols

The logical value assigned to the SETB symbol is substituted for the SETB symbol when it appears in the operand of an AIF instruction or another SETB instruction.

If a SETB symbol is used in the operand of a SETA instruction, or in arithmetic relations in the operands of AIF and SETB instructions, the binary values 1 (true) and 0 (false) are converted to the arithmetic values +1 and +0, respectively.

The following example illustrates these rules.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &NAME | MOVE | &TO,&FROM |
| | | LCLA | &A1 |
| | | LCLB | &B1,&B2 |
| | | LCLC | &C1 |
| 1 | &B1 | SETB | (L'&TO EQ4) |
| 2 | &B2 | SETB | (S'&TO EQ0) |
| 3 | &A1 | SETA | &B1 |
| 4 | &C1 | SETC | '&B2' |
| | | ST | 2,SAVEAREA |
| | | L | 2,&FROM&A1 |
| | | ST | 2,&TO&C1 |
| | | L | 2,SAVEAREA |
| | | MEND | |
| | FIELDA | DC | F'01' |
| | FIELDB | DC | DS3'12' |
| | HERE | MOVE | FIELDA,FIELDB |
| | HERE | ST | 2,SAVEAREA |
| | | L | 2,FIELDB1 |
| | | ST | 2,FIELDA0 |
| | | L | 2,SAVEAREA |

Because the operand of statement 1 is true, &B1 is assigned the binary value 1. Therefore, the arithmetic value +1 is substituted for &B1 in statement 3. Because the operand of statement 2 is false, &B2 is assigned the binary value 0. Therefore, the character value 0 is substituted for &B2 in statement 4.

The use of global SETB variable symbols is illustrated below.

### 4.17.4 Using Global SET Symbols

The examples in the previous discussion of the SET instructions used local variable symbols only.

The following examples illustrate the use of global and local SET symbols. Each example consists of two parts. The first part is an assembly language source program. The second part shows the statements that would be generated by the assembler after it processed the statements in the source program.

Example 1:

This example illustrates how the same SET symbol can be used to communicate:

1. values between statements in the same macro definitions, and

2. different values between statements outside macro definitions.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &NAME | LOAD | |
| 1 | | LCLA | &A |
| 2 | &NAME | LR | 15,&A |
| 3 | &A | SETA | &A+1 |
| | | MEND | |
| 4 | FIRST | LCLA | &A |
| | | LOAD | |
| 5 | | LR | 15,&A |
| | | LOAD | |
| 6 | | LR | 15,&A |
| | | END | FIRST |
| | FIRST | LR | 15,0 |
| | | LR | 15,0 |
| | | LR | 15,0 |
| | | LR | 15,0 |
| | | END | FIRST |

&A is defined as a local SETA symbol in a macro definition (statement 1) and outside macro definitions (statement 4). &A is used twice within macro definition (statement 2 and 3) and twice outside macro definitions (statement 5 and 6).

Since &A is a local SETA symbol in the macro definition and outside macro definitions, it is one SETA symbol in the macro definition, and another SETA symbol outside macro definitions. Therefore, statement 3 (which is in the macro definition) does not affect the value used for &A in statement 5 and 6 (which are outside macro definitions).

Example 2:

This example illustrates how the same SET symbol can be used to communicate:

1. values between statements in one macro definition, and

2. different values between statements in a different macro definition.

&A is defined as a local SETA symbol in two different macro definitions (statements 1 and 4). &A is used twice within each macro definition (statements 2, 3, 5 and 6).

Since &A is a local SETA symbol in each macro definition, it is one SETA symbol in one macro definition, and another SETA symbol in the other macro definition. Therefore, statement 3 (which is in one macro definition) does not affect the value used for &A in statement 5 (which is in the other macro definition). Similarly, statement 6 does not affect the value used for &A in statement 2.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &NAME | LOADA | |
| 1 | | LCLA | &A |
| 2 | &NAME | LR | 15,&A |
| 3 | &A | SETA | &A+1 |
| | | MEND | |
| | | MACRO | |
| | | LOADB | |
| 4 | | LCLA | &A |
| 5 | | LR | 15,&A |
| 6 | &A | SETA | &A+1 |
| | | MEND | |
| | FIRST | LOADA | |
| | | LOADB | |
| | | LOADA | |
| | | LOADB | |
| | | END | FIRST |
| | FIRST | LR | 15,0 |
| | | LR | 15,0 |
| | | LR | 15,0 |
| | | LR | 15,0 |
| | | END | FIRST |

Example 3:

This example illustrates how a SET symbol can be used to communicate values between statements that are part of two different macro definitions.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &NAME | LOADA | |
| 1 | | GBLA | &A |
| 2 | &NAME | LR | 15,&A |
| 3 | &A | SETA | &A+1 |
| | | MEND | |
| | | MACRO | |
| | | LOADB | |
| 4 | | GBLA | &A |
| 5 | | LR | 15,&A |
| 6 | &A | SETA | &A+1 |
| | | MEND | |
| | FIRST | LOADA | |
| | | LOADB | |
| | | LOADA | |
| | | LOADB | |
| | | END | FIRST |
| | FIRST | LR | 15,0 |
| | | LR | 15,1 |
| | | LR | 15,2 |
| | | LR | 15,3 |
| | | END | FIRST |

&A is defined as a global SETA symbol in two different macro definitions (statements 1 and 4). &A is used twice within each macro definition (statement 2, 3, 5 and 6).

Since &A is a global SETA symbol in each macro definition, it is the same SETA symbol in each macro definition. Therefore, statement 3 (which is in one macro definition) affects the value for &A in statement 5(which is in the other macro definition). Similarly, statement 6 affects the value used for &A in statement 2.

Example 4:

This example illustrates how the same SET symbol can be used to communicate:

1. values between statements in two different macro definitions, and

2. different values between statements outside macro definitions.

&A is defined as a global SETA symbol in two different macro definitions (statements 1 and 4), but it is defined as a local SETA symbol outside macro definitions (statement 7). &A is used twice within each macro definition and twice outside macro definitions (statements 2, 3, 5, 6, 8 and 9).

```
        Name     Operation   Operand
                 MACRO
        &NAME    LOADA
1                GBLA        &A
2       &NAME    LR          15,&A
3       &A       SETA        &A+1
                 MEND

                 MACRO
                 LOADB
4                GBLA        &A
5                LR          15,&A
6       &A       SETA        &A+1
                 MEND

7                LCLA        &A
        FIRST    LOADA
                 LOADB
8                LR          15,&A
                 LOADA
                 LOADB
9                LR          15,&A
                 END         FIRST
        FIRST    LR          15,0
                 LR          15,1
                 LR          15,0
                 LR          15,2
                 LR          15,3
                 LR          15,0
                 END         FIRST
```

Since &A is a global SETA symbol in each macro definition, it is the same SETA symbol in each macro definition. However, since &A is a local SETA symbol outside macro definitions, it is a different SETA symbol outside macro definitions.

Therefore, statement 3 (which is in one macro definition) affects the value used for &A in statement 5 (which is in the other macro definition), but it does not affect the value used for &A in statements 8 and 9 (which are outside macro definitions). Similarly, statement 6 affects the value used for &A in statement 2, but it does not affect the value used for &A in statement 8 and 9.

Example 5:

The next example illustrates the use of a global character variable symbol in conjunction with the system variable symbol &SYSNDX, to communicate between an inner and outer macro definition. OUTER1 is assumed to be the 106th macro instruction processed in the program.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | | INNER1 | |
| | | GBLC | &NDXNUM |
| 1 | A&SYSNDX | SR | 2,5 |
| | | CR | 2,5 |
| 2 | | BE | B&NDXNUM |
| 3 | | B | A&SYSNDX |
| | | MEND | |
| | | MACRO | |
| | &NAME | OUTER1 | |
| | | GBLC | &NDXNUM |
| 4 | &NDXNUM | SETC | '&SYSNDX' |
| | &NAME | SR | 2,4 |
| | | AR | 2,6 |
| 5 | | INNER1 | |
| 6 | B&SYSNDX | S | 2,=F'1000' |
| | | MEND | |
| 7 | ALPHA | OUTER1 | |
| 8 | BETA | OUTER1 | |
| | ALPHA | SR | 2,4 |
| | | AR | 2,6 |
| | A0107 | SR | 2,5 |
| | | CR | 2,5 |
| | | BE | B0106 |
| | | B | A0107 |
| | B0106 | S | 2,=F'1000' |
| | BETA | SR | 2,4 |
| | | AR | 2,6 |
| | A0109 | SR | 2,5 |
| | | CR | 2,5 |
| | | BE | B0108 |
| | | B | A0109 |
| | B0108 | S | 2,=F'1000' |

Statement 5 is the 107th macro instruction processed. Therefore, &SYSNDX is assigned the number 0107 for that macro instruction. The number 0107 is substituted for &SYSNDX when it is used in statements 1 and 3. The number 0106 is substituted for the global SETC symbol &NDXNUM in statement 2.

Statement 8 is the 108th macro instruction processed. Therefore, each occurrence of &SYSNDX is replaced by the number 0108. For example, statement 6 is used to create the unique name B0108.

When statement 5 is used to process the 108th macro instruction, statement 5 becomes the 109th macro instruction processed. Therefore, each occurrence of &SYSNDX is replaced by the number 0109. For example, statement 1 is used to create the unique name A0109.

Example 6:

Global SETB and SETC variable symbols are used in the following example to generate operation codes, symbols, and literals.

```
Name     Operation   Operand
         MACRO
&NAME    MAC1        &A,&B
         GBLA        &DUP
         GBLB        &BG8
         GBLC        &CG1
         LCLA        &AL1
&CG1     SETC        T&A
&AL1     SETA        L'&A
&BG8     SETB        (&B EQ")
&DUP     SETA        &AL1*4
&NAME    S&CG1       &A.(&AL1),=&CG1'01'
         MVC         FIELD&BG8(&AL1),&A
         MEND

CON1     DC          PL3'12'
CON2     DC          H'04'
BEGIN    MAC1        CON1,SWTCH
         MAC1        CON2
CON2     DC          H'04'
         END         BEGIN
```

```
CON1     DC          PL3'12'
CON2     DC          H'04'
BEGIN    SP          CON1(3),=P'01'
         MVC         FIELD0(3),CON1
         SH          CON2(2),=H'01'
         MVC         FIELD1(2),CON2
         END         BEGIN
```

### 4.17.5 Subscripted SET Symbols

Both global and local SET symbols may be defined as subscripted SET symbols inside and outside macro definitions. All SET symbols defined in the previous sections of this document are nonsubscripted SET symbols.

Subscripted SET symbols provide the programmer with a convenient way to use one SET symbol plus a subscript to refer to many arithmetic, binary, or character values.

A subscripted SET symbol consists of a SET symbol immediately followed by a subscript that is enclosed in parentheses. The subscript may be any arithmetic expression that is allowed in the operand of a SETA statement in the range of 1 to the specified dimension.

The following are valid subscripted SET symbols:

```
&READER(17)
&A23456(&S4)
&X4F2(25+&A2)
```

The following are invalid subscripted SET symbols:

```
&X4F2       (no subscript)
(25)        (no SET symbol)
&X4F2(25)   (subscript does not immediately follow SET symbol)
```

#### 4.17.5.1 Defining Subscripted SET Symbols

If the programmer wants to use a subscripted SET symbol, he must write the SET symbol in a GBLA, GBLB, GBLC, LCLA, LCLB, or LCLC instruction, immediately followed by an unsigned decimal integer enclosed in parentheses. The decimal integer, called a dimension, indicates the number of SET variables associated with the SET symbol. Every variable associated with a SET symbol is assigned an initial value that is the same as the initial value assigned to the corresponding type of nonsubscripted SET symbol.

If a subscripted SET symbol is defined as global, the same dimension must be used with the SET symbol each time it is defined as global.

The maximum dimension that can be used with a SETA, SETB, or SETC symbol is 2500. Dimension 1 is not permitted.

A subscripted SET symbol may be used only if the declaration was subscripted. A nonsubscripted SET symbol may be used only if the declaration had no subscript.

A period is not required after a nonsubscripted SET symbol for concatenation with a left parenthesis.

The following statements define the global SET symbols &SBOX, &WOBOX, and &PSW, and the local SET symbol &TSW. &SBOX has 50 arithmetic variables associated with it. &WBOX has 20 character variables, &PSW and &TSW each have 230 binary variables.

| Name | Operation | Operand |
|---|---|---|
| | GBLA | &SBOX(50) |
| | GBLC | &WBOX(20) |
| | GBLB | &PSW(230) |
| | LCLB | &TSW(230) |



### 4.17.5.2 Using Subscripted SET Symbols

After the programmer has associated a number of SET variables with a SET symbol, he may assign values to each of the variables and use them in other statements.

If the statement in the previous example were part of a macro definition, (and &A was defined as a SETA symbol in the same definition), the following statements could be part of the same macro definition.

| | Name | Operation | Operand |
|---|---|---|---|
| 1 | &A | SETA | 5 |
| 2 | &PSW(&A) | SETB | (6 LT 2) |
| 3 | &TSW(9) | SETB | (&PSW(&A)) |
| 4 | | A | 2,=F'&SBOX(45)' |
| 5 | | CLI | AREA,C'&WBOX(17)' |

Statement 1 assigns the arithmetic value 5 to the nonsubscripted SETA symbol &A. Statements 2 and 3 then assign the binary value 0 to subscripted SETB symbol &PSW(5) and &TSW(9), respectively. Statements 4 and 5 generate statements that add the value assigned to &SBOX(45) to general register 2, and compare the value assigned to &WBOX(17) to the value stored at AREA, respectively.

## 4.18 CONDITIONAL ASSEMBLY INSTRUCTIONS

The terms and expressions that are valid operands of the SET instructions may be used in conjunction with the conditional assembly instructions to indicate to the assembler the order in which statements should be processed.

There are 8 conditional assembly instructions which are described in this section.

| | | |
|---|---|---|
| AIF[B] | ANOP | NTRAC |
| AGO[B] | MEXIT | GSEQ |
| ACTR | MTRAC | |

All of the above statements may be used inside and outside macro definitions, except as noted in the text.

The AIF[B], AGO[B], ANOP and MEXIT instructions may be used to vary the sequence in which statements are assembled into a program. The programmer uses these statements to test the attributes of macro instruction operands and symbols defined in the program, and to test the current values of variable symbols, in order to determine which statements should be assembled. The ACTR instruction may be used to limit the number of AIF and AGO branches that are executed in an assembly.

MTRAC and NTRAC are used to trace the conditional logic of a macro definition or the source program at assembly time. MTRAC causes conditional statements to be listed with an indication, where applicable, of whether branching occurred. NTRAC negates the MTRAC function.

The GSEQ instruction is used to define conditional transfer points whose names may be generated, at assembly time, by variable symbols in the sequence symbol field of AIF[B] and AGO[B] statements.

Examples of all conditional assembly features are given throughout this section. A chart summarizing the elements which may be used in each instruction appears at the end of this section.

### 4.18.1 Sequence Symbols

Sequence symbols may be used in the name entry of most assembly language statements. They allow the programmer to refer to selected statements in the operands of the conditional instructions AIF[B] and AGO[B].

A sequence symbol may be used in any statement that does not contain a symbol or SET symbol except:

MACRO, LCLA, LCLB, LCLC, GBLA, GBLB, GBLC, GSEQ, ACTR, ICTL, ISEQ and START.

Sequence symbols may be written in a standard form or may be generated by variable symbols. The standard from of a sequence symbol consists of a period followed by one through seven letters and/or digits, the first of which must be a letter. A sequence symbol used in the name entry of a statement must be written in standard form.

Sequence symbols that appear in the operands of the conditional instructions may be written in standard or generated form. A generated sequence symbol consists of a period followed by

1. a variable symbol, or

2. a symbol followed by a variable symbol.

After all variables have been replaced by their current values, the resulting characters must form a standard sequence symbol.

If a sequence symbol is to be generated at assembly time, it must appear in the operand of a GSEQ statement. The GSEQ statement is described below.

Sequence symbols are local symbols. Thus, if the same symbol is used inside and outside a macro definition, or in two or more macro definitions, it is considered to be a different symbol in each case.

The following are valid standard form sequence symbols:

```
.READER      .A23456
.LOOP2       .X4F2
.N           .S4
```

The following are valid generated sequence symbols:

```
.&LOOP    .LOOP&AG1    .TAG&BG2
```

The following are invalid sequence symbols:

```
CARDAREA     (first character is not a period)
.246B        (first character after period is not a letter)
.AREA2456    (more than seven characters after period)
.BCD%84      (contains a special character other than initial period)
.IN AREA     (contains a blank)
```

If a sequence symbol appears in the name entry of a macro instruction, and the corresponding prototype statement contains a symbolic parameter in the name entry, the sequence symbol does not replace the symbolic parameter wherever it is used in the macro definition.

The following example illustrates this rule.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| 1 | &NAME | MOVE | &TO,&FROM |
| 2 | &NAME | ST | 2,SAVEAREA |
| | | L | 2,&FROM |
| | | ST | 2,&TO |
| | | L | 2,SAVEAREA |
| | | MEND | |
| 3 | .SYM | MOVE | FIELDA,FIELDB |
| 4 | | ST | 2,SAVEAREA |
| | | L | 2,FIELDB |
| | | ST | 2,FIELDA |
| | | L | 2,SAVEAREA |

If an AIF instruction appears in a macro definition, then the sequence symbol that is written or generated in the operand must appear in the name entry of a statement in that definition.

If an AIF instruction is used outside macro definitions, then the sequence symbol that is written or generated in the operand must appear in the name entry of a statement outside macro definitions.

Sequence symbols which may be generated by variable symbols must appear in a GSEQ statement in the macro definition or source program, depending on the placement of the conditional statement which references the symbol.

The following are valid operands of AIF instructions:

```
(&AREA+X'2D' GT 29).READER
(T'&P EQ 'F').THERE
(&AG1+3 GE 3*X'05').&TAG
(&BL1).LOOP&AL1
```

The following are invalid operands of AIF instructions:

```
(T'&ABC NE T'&XYZ)          (no sequence symbol)
.X4F2                       (no logical expression)
(T'&ABC NE T'&XYZ) .X4F2    blanks between logical expression and sequence
                            symbol
```

The following macro definition may be used to generate the statements needed to move a full-word fixed-point number from one storage area to another. The statements will be generated only if the type attribute of both storage areas is the letter F.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &N | MOVE | &T,&F |
| 1 | | AIF | (T'&T NE T'&F).END |
| 2 | | AIF | (T'&T NE 'F').END |
| 3 | &N | ST | 2,SAVEAREA |
| | | L | 2,&F |
| | | ST | 2,&T |
| | | L | 2,SAVEAREA |
| 4 | .END | MEND | |

The logical expression in the operand of statement 1 has the value true if the type attributes of the two macro instruction operands are not equal. If the type attributes are equal, the expression has the logical value false.

Therefore, if the type attributes are not equal, statement 4 (the statement named by the sequence symbol .END) is the next statement processed by the assembler. If the type attributes are equal, statement 2 (the next sequential statement) is processed.

The logical expression in the operand of statement 2 has the value true if the type attribute of the first macro operand is not the letter F. If the type attribute is the letter F, the expression has the logical value false. Therefore, if the type attribute is not the letter F, statement 4 (the statement named by the sequence symbol .END) is the next statement processed by the assembler. If the type attribute is the letter F, statement 3 (the next sequential statement) is processed.

Generated sequence symbols and AIF instructions used outside macro definitions are illustrated in the next example.

| Name | Operation | Operand |
|---|---|---|
| | START | |
| | GBLB | &B |
| | GSEQ | LOOP1,LOOP0 |
| | . | |
| | . | |
| | . | |
| | AIF | (&B EQ 1).LOOP&B |
| | AIF | (&B EQ 0).LOOP&B |
| | . | |
| | . | |
| | . | |
| .LOOP0 | DS | 5C |
| | . | |
| | . | |
| | . | |
| .LOOP1 | DS | 10C |

Additional illustrations follow the description of the AGO, ANOP and MEXIT instructions.

### 4.18.4 AGO - Unconditional Branch

The AGO(B) instruction is used to unconditionally alter the sequence in which source program or macro definition statements are processed by the assembler.

The format of this instruction is:

| Name | Operation | Operand |
|---|---|---|
| A sequence symbol or blank | AGO or AGOB | A sequence symbol |

The AGOB operation code is provided for TOS macro language compatibility. It may be used interchangeably with the AGO operation code.

The statement named by the sequence symbol in the operand is the next statement processed by the assembler.
The statement named by the sequence symbol may precede or follow the AGO instruction.
The sequence symbol in the operand entry may be in standard form or it may be generated. All rules that apply to sequence symbols in the AIF instruction operand are valid for the AGO instruction.

The following example illustrates the use of the AGO instructions:

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &NAME | MOVE | &T,&F |
| 1 | | AIF | (T'&T EQ 'F').FIRST |
| 2 | | AGO | .END |
| 3 | .FIRST | AIF | (T'&T NE T'&F).END |
| | &NAME | ST | 2,SAVEAREA |
| | | L | 2.&F |
| | | ST | 2,&T |
| | | L | 2,SAVEAREA |
| 4 | .END | MEND | |

Statement 1 is used to determine if the type attribute of the first macro instruction operand is the letter F. If the type attribute is the letter F, statement 3 is the next statement processed by the assembler. If the type attribute is not the letter F, statement 2 is the next statement processed by the assembler.

Statement 2 is used to indicate to the assembler that the next statement to be processed is statement 4 (the statement named by sequence symbol .END).

An AGO instruction is used below to branch around the definition of symbol A and one definition of symbol B. Not that attribute information for A will not be affected by the absence of A in the assembled program. The attributes of B, however, are undefined, although only one definition of B is generated.

| Name | Operation | Operand |
|---|---|---|
| | START | |
| | . | |
| | . | |
| | . | |
| | AGO | .L1 |
| A | DC | F'00' |
| .L1 | AIF | (T'A NE 'F').END |
| B | DC | H'00' |
| | AGO | .L2 |
| B | DC | X'00' |
| .L2 | AIF | (T'B NE 'H').END |
| | . | |
| | . | |
| | . | |
| .END | MEND | |

### 4.18.5 ANOP - Assembly No Operation

The ANOP instruction facilitates conditional and unconditional assembly time branching to statements whose name entry must be other than a sequence symbol.

The format of this instruction is:

| Name | Operation | Operand |
|---|---|---|
| A sequence symbol | ANOP | Blank |

If the programmer wants to use an AIF or AGO instruction to branch to another statement, a sequence symbol must be placed in the name entry of the statement to which the branch is to take place. However, if the programmer has already entered a symbol or variable symbol in the name entry of that statement, a sequence symbol cannot be placed in the name entry. Instead, the programmer must place an ANOP instruction before the statement and then branch to the ANOP instruction. This has the same effect as branching to the statement immediately after the ANOP instruction.

The following example illustrates the use of the ANOP instruction.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | &NAME | MOVE | &T,&F |
| | | LCLC | &TYPE |
| 1 | | AIF | (T'&T EQ 'F').FTYPE |
| 2 | &TYPE | SETC | 'E' |
| 3 | .FTYP | ANOP | |
| 4 | &NAME | ST&TAPE | 2,SAVEAREA |
| | | L&TYPE | 2,&F |
| | | ST&TYPE | 2,&F |
| | | L&TYPE | 2,SAVEAREA |
| | | MEND | |

Statement 1 is used to determine if the type attribute of the first macro instruction operand is the letter F. If the type attribute is not the letter F, statement 2 is the next statement processed by the assembler. If the type attribute is the letter F, statement 4 should be processed next. However, since there is a variable symbol (&NAME) in the name field of statement 4, the required sequence symbol (.FTYPE) cannot be placed in the name field. Therefore, an ANOP instruction (statement 3) must be placed before statement 4. Then, if the type attribute of the first operand is the letter F, the next statement processed by the assembler is the statement named by sequence symbol .FTYPE. The value of &TYPE retains its initial null character value because the SETC instruction is not processed. Since .FTYPE names an ANOP instruction, the next statement processed by the assembler is statement 4, the statement following the ANOP instruction.

The example below illustrates the use of the ANOP statement in conjunction with forward and backward conditional transfers.

```
     Name      Operation     Operand
                MACRO
     &NAME      SAVE         &A,&B
1               AIF          ('&A'EQ'').L2
7    .L1        ANOP
2    &NAME      MVC          &A.(4),&B
3               AGO          .L3
4    .L2        ANOP
5    &A         SETC         'SAVE'
     SAVE       DC           A(0)
6               AGO          .L1
     .L3        MEND
```

Statement 1 is used to test for the presence of the first operand. If the operand is present, its value is used in statement 2, and the macro processing is terminated by statement 3. If the operand is omitted, the SETC statement in line 5 is used to create a value for &A. Because the SETC statement requires a variable symbol in the name entry, the ANOP statement in line 4 is necessary for branching. Statement 6 results in a transfer back to line 7. The instruction in statement 2 is then generated with the new value of &A.

### 4.18.6 MEXIT - Macro Definition Exit

The MEXIT instruction is used to indicate to the assembler that it should terminate processing of the current macro definition.

The format of the MEXIT instruction is:

| Name | Operation | Operand |
|---|---|---|
| A sequence symbol or blank | MEXIT | Blank |

The MEXIT instruction may only be used in a macro definition. It should not be confused with the MEND statement which must be the last statement of every macro definition, including those that contain one or more MEXIT instructions.

When the assembler detects a MEXIT instruction within a macro definition, the next statement processed will be the statement that immediately follows the corresponding macro instruction. If the macro instruction appeared in the source program, the next source program statement will be processed. If the macro instruction appeared as a model line in an outer level macro definition, the next model statement of that macro definition will be processed.

Conditional instructions, MEXIT and MNOTE statements, and the &SYSLIST feature are used in the example below to test the attributes of macro instruction operands.

| | Name | Operation | Operand |
|---|---|---|---|
| | | MACRO | |
| | | TEST | &A |
| | | LCLA | &AL1 |
| | | LCLC | &CL1 |
| | .REPEAT | DS | OC |
| 1 | &AL1 | SETA | &AL1+1 |
| 2 | | AIF | ('&SYSLIST(&AL1)'EQ'').OUT |
| 3 | | AIF | (T'&SYSLIST(&AL1) EQ 'N').L1 |
| | &CL1 | SETC | T'&SYSLIST(&AL1) |
| 4 | | MNOTE | 1,'PARAM &AL1 IS NOT SELF DEFINING' |
| 5 | | MNOTE | 2,'TYPE IS &CL1' |
| 6 | | AIF | (&AL1 NE 3).REPEAT |
| | .OUT | MEXIT | |
| 7 | .L1 | MVI | SYM,&SYSLIST(&AL1) |
| | | AIF | (&AL1 NE 3).REPEAT |
| | | MEND | |

Statement 1 increments an arithmetic local variable symbol that is used as a parameter counter. In statement 2, a test is made to determine if a corresponding operand was supplied. If not, the macro processing is terminated by a branch to the MEXIT instruction. Otherwise, a test of the type attribute is made in statement 3. If the type attribute is self-defining, line 7 is processed and a move of the parameter to SYM is generated. If the expression in line 3 is false, the MNOTE statements in statements 4 and 5 are used to warn the programmer that the desired conditions are not satisfied. The tests are repeated until the arithmetic expression in statement 7 is false; i.e., &AL1 equals 3; or until an operand is omitted.

### 4.18.7 ACTR - Conditional Assembly Loop Counter

The ACTR instruction is used to limit the number of AGO and AIF branches executed within the source program.

A separate ACTR statement may be used in each macro definition and in the main program. These counters are independent.

The format of this instruction is:

| Name | Operation | Operand |
|---|---|---|
| Not used | ACTR | Any valid SETA expression |

This statement causes a counter to be set to the value in its operand. Each time an AGO or AIF branch is executed, the counter is decremented by one. If the count is zero before decrementing, the assembler takes one of two actions:

1. If a macro definition is being processed, the processing of it and any macros above it in a nest is terminated, and the next statement in the main portion of the program is processed.

2. If the main portion of the program is being processed, conditional assembly is terminated, and the portion of the program generated so far is assembled.

If an ACTR statement is not given, the value of the counter is assumed to be 1200.

### 4.18.8 MTRAC - Macro Trace

The MTRAC instruction allows the programmer to determine the effective conditional transfers within a macro definition or the source program. If MTRAC is specified, each conditional instruction (see section 4.18) is printed on the assembly listing and a "Y" or "N" designation in column 80 indicates whether or not the branch was performed. The definition of the local and global SET parameters (see section 4.14.2) is likewise printed on the assembly listing if MTRAC is specified.

Every current value of an executed SET instruction (SETA, SETB, SETC) and every ACTR operand is printed in columns 73-80.

a) **SETA parameters and ACTR operands:** shown in columns 73-80 as decimal numbers with leading zeros; for negative values, rather than indicating a sign, the last digit is shown as follows:

| | |
|---|---|
| 0 is not printed | 5 = N |
| 1 = J | 6 = O |
| 2 = K | 7 = P |
| 3 = L | 8 = Q |
| 4 = M | 9 = R |

For numbers that have more than 8 digits, an asterisk (*) is printed in column 73, followed by the 7 least-significant digits.

b) **SETB parameters:** shown as a single character in column 80; 'T' = true or (1); 'F' = false or (0).

c) **SETC parameters or symbolic parameters in SETC statements:** shown as a sequence of up to eight characters, beginning with column 73, space-filled (omitted parameter is printed as ␣␣NULL␣␣). If the sequence has more than 8 characters, then only the first 7 characters are printed and an asterisk (*) is placed in column 80.

The format of the MTRAC instruction is:

| Name | Operation | Operand |
|---|---|---|
| Sequence symbol or blank | MTRAC | Blank |

The MTRAC instruction may be used inside and outside macro definitions. If the NOGEN option is on, MTRAC has no effect.



Note

If the MTRAC function is activated in a higher-level library macro, it is applicable to lower-level macros even if these were called in the program before the MTRAC instruction proper.

### 4.18.9 NTRAC - No Trace

The NTRAC instruction negates the MTRAC function. The format of the NTRAC instruction is identical to the format of MTRAC instruction.
If MTRAC is not specified, any NTRAC processed has no effect.

### 4.18.10 Conditional Assembly Elements

The following chart summarizes the elements that can be used in each conditional assembly instruction. Each row in this chart indicates which elements can be used in a single conditional assembly instruction. Each column is used to indicate the conditional assembly instructions in which a particular element can be used.

The intersection of a column and a row indicates whether an element can be used in an instruction, and if so, in what fields of the instruction the element can be used. For example, the intersection of the first row and the first column of the chart indicates that symbolic parameters can be used in the operand field of SETA instructions.

| | Variable symbols | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | S.P. | SET symbols | | | System variable symbols | | | Attributes | | | | | | |
| | | SETA | SETB | SETC | &SYSNDX | &SYSECT | &SYSLIST | T' | L' | S' | I' | K' | N' | S.S. |
| SETA | 3 0 ) | N,0 | 0 | 5 0 ) | 0 | | 3 0 ) | | 0 | 0 | 0 | 0 | 0 | |
| SETB | 4 0 ) | 4 0 ) | N,0 | 4 0 ) | 4 0 ) | 1 0 ) | 4 0 ) | 1 0 ) | 2 0 ) | 2 0 ) | 2 0 ) | 2 0 ) | 2 0 ) | |
| SETC | N,0 | 0 | 0 | N,0 | 0 | 0 | 0 | 0 | | | | | | |
| AIF | 4 0 ) | 4 0 ) | 0 | 4 0 ) | 4 0 ) | 1 0 ) | 4 0 ) | 1 0 ) | 2 0 ) | 2 0 ) | 2 0 ) | 2 0 ) | 2 0 ) | N,0 |
| AGO | | | | | | | | | | | | | | N,0 |
| ANOP | | | | | | | | | | | | | | N |
| ACTR | 2 0 ) | 0 | 0 | 5 0 ) | 0 | | 3 0 ) | | 0 | 0 | 0 | 0 | 0 | |

Fig. 4-1: Conditional Assembly Elements

1) Only in character relations
2) Only in arithmetic relations
3) Only if value is a self-defining term
4) Only in character or arithmetic relations
5) Only if value is a decimal self-defining term

**Abbreviations**

N = is Name
O = is Operand
S.P. = is Symbolic Parameter
L' = is Length Attribute
S' = is Scaling Attribute
I' = is Integer Attribute
K' = is Count Attribute
N' = is Number Attribute
S.S. = is Sequence Symbol

# 5 POST-ASSEMBLY DIAGNOSTIC ROUTINE (ADIAG)

## 5.1 USAGE

The Post-Assembly Diagnostic Routine (ADIAG) is a conversational routine which has been specially generated for version 29.1 of the assembler, and which performs the following basic functions:

- output of diagnostic information about a preceding assembly
- input/output mapping in conversational mode
- COMOPT-controlled, implicit start via the assembler when a particular error severity is reached
- explicit start via a /EXEC command
- correction of the source program in conversational mode with the aid of the file editor EDT
- restart of the assembler
- matching of the diagnostic program to the reorganization of the assembler flags

It is a prerequisite of using the ADIAG ≥ V29.1 that the diagnostic file has been generated by the assembler ≥ V29.1, i.e. diagnostic files generated using previous assembler versions cannot be evaluated using ADIAG V29.1; similarly previous ADIAG versions cannot evaluate the diagnostic files of the assembler V29.1.

The diagnostic file is generated using one of the following entries:

- *COMOPT {SAVLST / ADIAG=n} during assembly
- /PARAM SAVLST=ALL prior to calling the assembler

The diagnostic program may be initiated either explicitly by means of an /EXEC command or implicitly by means of the assembler.
When an implicit start is selected the $ADIAG maintained in the system catalog is always initiated. This start is initialized by entering *COMOPT ADIAG=n, and takes effect if a particular error severity is determined during the assembly.
When *COMOPT ADIAG=0 is entered, the $ADIAG is initiated irrespective of any errors which might arise.

The diagnostic program can be initiated either in conversational mode or in batch mode; however, not all ADIAG commands can be processed in batch mode.

## 5.2 DEFINITIONS

**Diagnostic file**

An ISAM file generated by the assembler and containing the diagnostic information.

**ADIAG command**

Command to the diagnostic program to perform certain services.

**Severity code**

All errors determined by the assembler during assembly of a program are assigned one of the severity codes described below:

| Code | Severity code | Description |
|---|---|---|
| WAR | 0 | WARNING;<br>Warning - successful program run possible |
| SIG | 1 | SIGNIFICANT ERROR:<br>Error - illegal program run possible |
| SER | 2 | SERIOUS ERROR;<br>Serious error - program run not possible |
| FAT | 3 | FATAL ERROR;<br>Fatal error - assembly terminated; diagnostic file incomplete |

**Error type**

One of the letters A-Z with which each error code begins.

**Error code**

Error type followed by one or two digits used to identify an error.

**Error text**

Verbal description of an error code.

## 5.3 STRUCTURE AND USE OF THE DIAGNOSTIC FILE

### 5.3.1 Structure

The Diagnostic File is created (optionally) at assembly time. The file will contain all data that appears in an assembly listing, plus some additional information for the diagnostic program.
The Diagnostic File is an ISAM file. It is generated optionally using the corresponding COMOPTs SAVLST or ADIAG=n. If the COMOPT ADIAG=n is entered, this implies a SAVLST entry. If the PARAM command /PARAM SAVLST=ALL is specified prior to calling the assembler, a diagnostic file containing the entire assembly listing will likewise be generated.



### 5.3.2 Naming the Diagnostic File

The assembler generates diagnostic files with the following names:

Entry: *COMOPT {SAVLST / ADIAG=n} or /PARAM SAVLST=ALL

Filename: SAVLST.ASSEMB.<name of 1st CSECT> with file link name SAVLINK

SAVLST.ASSEMB.<tsn>
if the 1st CSECT is unnamed

If the diagnostic file is to be given a different name, a FILE command with the file link name SAVLINK must be entered before calling the assembler.

Note

If, in a job, the assembler is started more than once by means of the SAVLST option, a /RELEASE SAVLINK must be issued before every restart of the assembler to avoid overwriting.

## 5.4 START OF THE DIAGNOSTIC ROUTINE

The ADIAG can be initiated either explicitly by means of an /EXEC command or implicitly by means of the assembler.

### 5.4.1 Explicit Start

The ADIAG is started by means of an /EXEC command. The mechanisms triggered by this start correspond to those for an implicit start.

### 5.4.2 Implicit Start

The COMOPT ADIAG=n permits the user to initiate the diagnostic routine by means of the assembler in accordance with the highest severity code encountered during the assembly. The start is performed only after *END HALT or *HALT is entered.
The meaning of the value n is as follows:

| n | ADIAG start |
|---|---|
| 0 | Following the assembly, irrespective of the assembly result |
| 1 | On encountering the WAR severity code (warning) or a higher severity code |
| 2 | On encountering the SIG severity code (significant error) or a higher severity code |
| 3 | On encountering the SER severity code (serious error) or a higher severity code |

If a fatal error is encountered the ADIAG is not initiated implicitly by the assembler.

Following initiation the diagnostic program attempts to open a diagnostic file using the file link name SAVLINK. If this is not possible, it is up to the user to specify a diagnostic file by means of the ADIAG command OPEN. When the file has been opened, the ADIAG evaluates in the diagnostic file the information that is beyond the scope of the assembly listing, outputs this information and waits for further ADIAG commands to be entered.

## 5.5 COMMAND INTERRUPTION

If the user is requested to enter an ADIAG command, he can branch to system mode and enter BS2000 commands by pressing the BREAK key.
The interrupted ADIAG run can be continued by entering the RESUME command.

## 5.6 ADIAG COMMANDS



IADAG is controlled by means of commands. These are read in interactive mode with the aid of the WRTRD macro. In batch mode and by setting program switch 1, the commands are read from SYSDTA. In interactive mode, the commands must always be entered on the first line (command line) of the screen.

General format:

<commandname> [<operand>[,<operand>...]]

### 5.6.1 Overview of ADIAG Commands

| Command | Function |
|---|---|
| CDT | Calls the file editor EDT for correcting source lines |
| COMOPT | Lists the COMOPTs entered during the assembly (with modification option) |
| DISPLAY | Outputs error causes and numbers of affected statements on the data display terminal |
| END | Terminates execution of the Diagnostic Program |
| HELP | Lists and explains ADIAG Commands |
| LIST | As DISPLAY, but outputs on the basis of SYSLST |
| OPEN | Opens a diagnostic file; assembly result in abbreviated form |
| PRINT | Lists statements |
| RERUN | Starts the assembler using the valid COMOPTs |
| SYSTEM | Executes a system command |
| TAGS | Produces a list of all symbols that are multiply defined and/or undefined |
| XREF | Lists cross-reference data |

## 5.6.2 Command Definitions

### 5.6.2.1 CDT Command

Format: C[DT] [<parameter>[,<parameter>[,<parameter>]]]

Parameter:

```
                 ⎧<errortype>     ⎫
<parameter>::=   ⎨<errorcode>     ⎬
                 ⎩<statementno.>  ⎭
```

**Function:**

The file editor EDT is initiated as a subroutine, and the corresponding source program file (or corrections file) is opened. The statement with the specified number or all statements with the identified error must be corrected.

**CDT Command without parameter**

If the command is entered without a parameter, it is up to the user to search for invalid source lines and to correct these. The EDT is started and the source file opened in full screen mode. It can then be processed using the EDT options. After you have saved the file, the EDT command H[ALT] is used to return to the ADIAG. Further parameter-controlled processing of the source using the CDT command is then generally no longer possible (see Notes).

**CDT command with parameter**

The program environment (the next 4 source lines) is output additionally to each invalid source line. The invalid line can then be corrected using the standard EDT procedure.
The next invalid source line is then displayed together with its program environment.
This procedure is repeated until all desired corrections have been processed.
If no corrections are made, @RETURN is used to scroll to the next line to be corrected.
When the EDT commands @PRINT or @LIST have been executed there is no scrolling, but rather the source lines previously output is output again.

Restriction:

When this procedure is used only the following EDT commands are permitted:
@RETURN, @PRINT, @LIST, @SAVE and @WRITE.

Notes:

- Source statements which originate a 'MULTIPLY DEFINED SYMBOL' flag are processed at the end of automatic scrolling.
- It is up to the user to eliminate 'UNDEFINED SYMBOL' flags following automatic scrolling.
- If a corrections file was involved in the preceding assembly, then first the source file and then the corrections file (if affected) are processed by means of the above procedure.
- The assembler determines the assignment between the statement number and the source line number during the assembly. This assignment may not be destroyed during automatic scrolling, e.g. source lines may only be deleted following termination of scrolling by the user.



Attention

The CDT command is not possible in batch mode.

The processing of library members with the CDT command is not possible.

#### 5.6.2.2 COMOPT Command

Format: CO[MOPT]

**Function:**

The COMOPTs set when the diagnostic file is created are displayed on the data display terminal; they can be redefined for a new assembly initiated by means of the RERUN command. This is achieved by modifying the screen display directly, then entering the next ADIAG command in the command line.

Note:

This function is not possible in batch mode.

### 5.6.2.3 DISPLAY Command

Format: D[ISPLAY] [<parameter>]

Parameter:

```
                 ⎡WAR                                ⎤
                 ⎢SIG                                ⎥
                 ⎢SER                                ⎥
<parameter>::=   ⎢MNOTES                             ⎬
                 ⎢<statementno.> [-<statementno.>]   ⎥
                 ⎢<errortype>                        ⎥
                 ⎣<errorcode>                        ⎦
```

**Function:**

The error causes with reference to statement numbers are output on the data display terminal on condition that the assembly is not aborted (see "Special case").

- Non-parametrized statement:

  Listing of all flags and MNOTES identified, classified according to severity code, error code with error text and references to the affected statement numbers.

- WAR:

  As above, but only the "Warning" severity code.

- SIG:

  As above, but only the "Error" severity code.

- SER:

  As above, but only the "Serious error" severity code.

- MNOTES:

  Listing of all MNOTES with assigned severity code and text, followed by the statement number.

- <statementno.> [-<statementno.>]

  The invalid statement or statements (no MNOTES) encountered in the specified number range are each output on the data display terminal with a succeeding error code and error text.

- <errortype>

  Listing of the errors of an error type, followed by a precise error code and error text and the affected statement numbers.

- <errorcode>

  As above, but only the specified error code.

- Scroll function

  If the output is spread over several screens, it is possible to scroll to the next screen by means of a null input.

Special case: Assembly aborted

If the assembly is aborted, it is not possible to generate a complete SAVLST. Two different outputs are then possible depending on the cause of the abortion:

1. Abortion with error severity 3 (i.e. continuation of assembler run not possible):

   The error text is output with notes referring to the possible causes. In such cases all parameters in the DISPLAY statement are ignored.

2. Controlled abortion by entering the highest error severity:

   All statements so far flagged are output similarly to an ERRFIL. These must however include at least the statement triggering the abortion (e.g. MNOTE with SEV-CODE=255).
   In such cases DISPLAY parameters are accepted with restrictions (statement references are always excluded).

### 5.6.2.4 END Command

Format: E[ND] [L]

**Function:**

The diagnostic program is terminated. If the supplement 'L' is added, the assembly listing is output via SYSLST. Files that are open are closed.

### 5.6.2.5 HELP Command

Format: H[ELP] [<command>]

**Function:**

Listing of all ADIAG commands or description of selected ADIAG commands on the data display terminal.

Note:

This function is not possible in batch mode.

### 5.6.2.6 LIST Command

Format: L[IST] [<parameter>]

Parameter:

See 5.6.2.3, DISPLAY Command

**Function:**

Similar to DISPLAY command, but output to SYSLST.

### 5.6.2.7 OPEN Command

Format: O[PEN] <filename>

Parameter:

```
                 ⎡SAVLST.ASSEMB.<CSECT-name>                        ⎤
<filename>:: = ⎨SAVLST.ASSEMB.<tsn> with file link name SAVLNK⎬
                 ⎣file name                                         ⎦
```

**Function:**

The diagnostic file specified is opened, and any previously opened diagnostic file with a different name is closed beforehand. The assembly listing is output to the appropriate output medium in abbreviated form.



### 5.6.2.8 PRINT Command

Format:

```
          ⎡<statementno.>[-<statementno.>]⎤
P[RINT] [⎨                                ⎬][,L][,S]
          ⎣<symbol.>[-<symbol>]           ⎦
```

**Function:**

Lists a specified statement or a specified range of statements as they would appear on the assembly listing. If no range is specified, all statements are listed. However the rightmost characters of any line extending beyond 80 columns are truncated on the data display terminal.
Specifying 'S' causes only the following information of a line to be output:

- location counter
- statement number
- source statement.

Specifying 'L' triggers an additional, complete output of the printed lines as per SYSLST. The error codes and error texts are inserted after the affected statements.

If the output is spread over several screens, it is possible to scroll to the next screen by means of a null input.

Note:

Not possible if the assembly is aborted.

### 5.6.2.9 RERUN Command

Format: R[ERUN]

**Function:**

The assembler is restarted by means of the COMOPTs as set on creation of the diagnostic file, or as reset by means of a preceding COMOPT command.

Note:

This function is not possible in batch mode.

### 5.6.2.10 SYSTEM Command

Format: S[YSTEM]<parameter>

Parameter:

<parameter>::='system-command'

**Function:**

The system command, enclosed in apostrophes, can be specified with or without slash. It is executed right away and, subsequently, the ADIAG run is continued, as far as the preceding system command allows.

All commands are allowed if they are called by means of the CMD macro (see "Executive Macros" reference manual).

Note:

ADIAG remains loaded and open files are not closed throughout command execution.

### 5.6.2.11 TAGS Command

Format: T[AGS] [<type>[,<type>]][,X[REF]

Parameter:

<type>::={M, U}

Default value: M,U

**Function:**

Lists all symbols that are undefined (U) and/or multiply defined (M). Cross-reference data is also listed if XREF specified.
If the output is spread over several screens, it is possible to scroll to the next screen by means of a null input.

Note:

This function is not possible in batch mode or if the assembly is aborted.

### 5.6.2.12 XREF Command



Format: X[REF] <parameter>[,<parameter>[,<parameter>]]

Parameter:

```
                 [<symbol>[-<specification>]]
<parameter>::=   {*<macroname>               }
                 [<literal>                  ]

                 [*]
                 [A]
                 [R]
<specification>::=[ ]
                 [W]
                 [E]
                 [O]
```

**Function:**

The cross-reference data for the symbols, macro names or literals specified is displayed on the data display terminal.
If *COMOPT ATXREF was specified during the assembly, it is possible to request certain cross-references for symbols:

*: Definition line
A: Address accesses
R: Non-write accesses via commands
W: Write accesses
E: Symbol for EQU-/ORG statement
O: Other accesses via assembly statements.

If the output is spread over several screens, it is possible to scroll to the next screen by means of a null input.

Note:

Not possible if the assembly is aborted.

### 5.6.3 Formatted Screen I/O

**Example: Basic structure of ADIAG formats**

```
CMD: aaaaaaaa                    A D I A G                    VERSION: V29.1C

                    BASIC STRUCTURE OF ADIAG FORMATS

GIVE ADIAG CMD: ____________________________________________________________
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  PAGE:aa
```

**Example: DISPLAY Command**

```
CMD: DISPLAY                A D I A G                        VERSION: 29.1C

CLASS FLAG  MESSAGE AND STATEMENT NUMBERS
SER   D      NONE
SIG   D1    INVALID CONSTANT TYPE
            00006
      D4    QUOTES NOT PAIRED OR ILLEGAL TERMINATION OF QUOTE STRING
            00002  00003  00004
WAR   D      NONE

GIVE ADIAG CMD :
 END OF OUTPUT                                                PAGE: 1
```

**Example: XREF Command**

```
CMD: XREF                          A D I A G                          VERSION: 29.1C

SYMBOL     LEN  VALUE  DEFN    REFERENCES
ERCSW    00001 001BF5 02791  03203W 03218R 03415W 03481W 03580R 03581W 05288W
                             05397R 06841W 06886R 06983R 06990R 06997R
FLAGTYP  00004 0008F0        05535W 05563W 05653W 07053W 07081W 07138W
ON       00001 000001 00605  00807R 00896R 01334R 01675R 02326R 03115R 03203R
                             03218R 03397R 03443R 03481R 03558R 03580R 03592R
                             03681R 04278R 04712R 05008R 05248R 05251R 05318R
                             05324R 05397R 05399R 05408R 05450R 05528R 05568R
                             05569R 05608R 05609R 05774R 05880R 06210R 06597R
                             06804R 06886R 06888R 06895R 07330R 07746R 07773R

GIVE ADIAG CMD :
 END OF OUTPUT                                                          PAGE: 1
```

# 6 ASSEMBLING A PROGRAM

The input of source programs, the output of listings and the internal assembly process can be controlled by options. The following possibilities are provided for entering these options:

- *COMOPT statements
- PARAMETER command
- Explicit input.

## 6.1 *COMOPT STATEMENTS



*COMOPT statements are read from SYSDTA

- as soon as the assembler has been loaded
- during assembly from SYSDTA after termination of the source program
- during assembly from file or library after the EOF condition has been reached.

A *COMOPT statement has a maximum length of 80 characters. It begins with COMOPT followed by one or more options separated by commas. Continuation lines are not possible. The input of *COMOPT statements is terminated by *END.

Any redefinitions of the logical system file SYSDTA are made immediately after the *END statement. Subsequently, *STARTC cards may be read from SYSDTA as before if the assignment expects any source program lines from SYSDTA.

The *COMOPT statements are listed completely as soon as at least one *COMOPT statement is used. Errors are output to SYSLST and SYSOUT and can be corrected with the aid of another *COMOPT statement.

As the assembler, during the input of *COMOPTs via SYSCMD, and the input of the source program, from file or library after termination of the assembly, will again call *COMOPTs for the next assembly, two further control statements are available in addition to the EOF condition in order to terminate the assembler:

- The HALT operand in the *END statement permits the assembler to be terminated after the next assembly.
- The *HALT statement instead of an *COMOPT or *END statement causes the assembler to be terminated right away.
- If, inadvertently, invalid information was entered instead of the first *COMOPT command, the assembler will interpret it as the first source line. By entering ␣END you can start the assembly of this invalid input, and then enter the correct options for the next assembly.

### 6.1.1 Summary of all Options

| *COMOPT | Meaning |
|---|---|
| ADIAG=n | A diagnostic file is generated (see COMOPT SAVLST). Following the assembly the $ADIAG routine is started implicitly when errors with the value n (see section 5.4.2) or MNOTES with a corresponding severity code are encountered.<br>0 ≤ n ≤ 3 |
| ALTLIB[n] | Assigns a macro library or the n-th macro library ( 2 ≤ n ≤ 5) |
| NOALTLIB[n] | Default assignment |
| ATXREF | The references in the cross reference listing are displayed together with an attribute that refers to the mode of access.<br>W Write access<br>R Non-write access by instructions<br>A Address access<br>E EQU/ORG statements<br>Blank Other assembly statements<br><br>ATXREF is only effective in conjunction with the INSTR=SET1 option. |
| NOATXREF | Default assignment. Not attribute XREF. |
| COPYMAC | Deactivates the COPY statement as a mnemonic assembly code. Any present mnemonic COPY Opcode will be interpreted as a macro instruction. Corresponds to the COPY OPSYN statement at the beginning of the program. |
| NOCOPYMAC | Default assignment |
| DUET | Allows TRANSDATA 960 instructions |
| NODUET | Default assignment |
| ERR=n | The assembly is terminated with TERMJ if more than n errors.<br>0 ≤ n ≤ 255 |
| ERR=Sm | The assembly is terminated with TERMJ if errors with the severity m or MNOTES with a class m severity code are encountered.<br>0 ≤ m ≤ 3 |

| *COMOPT | Meaning |
|---|---|
| ERRFIL | Outputs error listing to an error file having the link name ERRLINK. If the file is not known at assembly time, it is created with the name<br><br>ERRFIL.ASSEMB.{<TSN> / <CSECT name>}<br><br>Note:: If ERRFIL and SAVLST are specified simultaneously, ERRFIL is ignored. |
| NOERRFIL | Default assignment |
| ERRPR=n | Only those error flags with a severity lower than n are evaluated for the end message of the assembler and for the COMOPT ERR.<br>0 ≤ n ≤ 3 Default value: n=1 |
| FLGLST | Where statements contain error flags the error code and the corresponding description are printed after the line. |
| NOFLGLST | Where statements contain error flags the error types (max. 3) are printed in the left-hand margin of the line. |
| HWTST | For special tests the CCW flag may be transferred unmodified (unabbreviated). However, this occurence itself will be flagged. |
| NOHWTST | No unmodified transfer of the CCW flag byte. |
| INSTR={SET1 / SET2} | Specifies the instruction set to be generated.<br>SET1 Instruction set of systems 7.500 or 7.700 (see Appendix A.6)<br><br>SET2 Instruction set of system 7800 (see Appendix A.7) |
| ISD | The assembler outputs ISD cards to the object module. |
| NOISD | Default assignment |

| *COMOPT | Meaning |
|---|---|
| LINECNT=n | Line/print page control, including header line (15 ≤ n ≤ 255).<br>Default assignment: n = 60 |
| LIST | Default assignment. The assembly listing is output to SYSLIST. |
| NOLIST | Only the invalid statements are displayed. |
| MCALL | Causes inner macros of library macros to be read only if they are defined in a MCALL statement.<br>Only these macros can be expanded. |
| NOMCALL | Default assignment |
| MDIAG | Macro definitions of library macros are to be printed; generation is inhibited. |
| NOMDIAG | Default assignment |
| MLPRNT | The macro identification line consisting of version number, creation date and link name of the macro library is output to the assembly listing. The version number consists of blanks if the macro was entered in the macro library by the MLU utility routine. |
| NOMLPRNT | Default assignment |
| MODULE=<br>specification | Specifies where the object module is to be output to. If this option is not used, the object module is output to the EAM file.<br>For a full description of this option see section 6.1.3. |
| NDLIST | Generates laser printer oriented assembly listing. |
| NONDLIST | Default assignment |
| OUTPUT=filename | Only in combination with *COMOPT UPD. The corrected source program is output to the specified file. |
| PROCOM | Output of program complexity metrics of the source program (see Appendix A.10). |
| NOPROCOM | Default assignment. No output of program complexity metrics. |
| PRTALL | Generates a complete assembly listing. The options of the PRINT statement, NOGEN, OFF, NOCOPY, NOREF and xOFF, are suppressed. |
| NOPRTALL | Default assignment |

| *COMOPT | Meaning |
|---|---|
| PRTIT | The effect of the TITLE statements generated by means of macros is maintained, even if printing of the TITLE the TITLE statements is suppressed by means of PRINT NOGEN. |
| NOPRTIT | The effect of the TITLE statements generated by means of macros is suppressed by means of PRINT NOGEN. |
| PRTOFF= {n / X1[;X2]...[;X5]} | The statements generated by means of macros are either printed or not printed, depending on the macro stage and the first character of the macro name.<br><br>n Statements generated by means of macros are not printed as of the n-th macro stage.<br>1 ≤ n ≤ 250<br><br>X1[;X2]...[;X5]<br>Statements generated by means of macros are never printed if the first character of the macro name is specified in the list X1 to X5 (irrespective of any macro stage set by means of PRTOFF=n). This list may contain up to five first characters of macro names; all macros whose name begins with one of these characters are affected. |
| SAVLST | The assembly listing is output to a diagnostic file having the link name SAVLINK. If the file is unknown at assembly time, it is created with the name<br><br>SAVLST.ASSEMB.{<TSN> / <CSECT name>}<br><br>Note:<br>If SAVLST and ERRFIL are specified simultaneously, ERRFIL is ignored.<br><br>If, in a job, the assembler is started more than once by means of the SAVLST option, a /RELEASE SAVLINK must be issued before every restart of the assembler so as to avoid overwriting. |
| NOSAVLST | Default assignment |
| SEQ=(number [,length[,id]]) | Specifications for the identification field in the assembly listing or in the corrected source program (see option OUTPUT= ).<br>number = Initial numbering with an increment of 100 for the identification field. Leading zeros may be omitted.<br>length = Length of numbering, starting from the right in the identification field (4 ≤ length ≤8).<br>id = Identifier to be copied into the identification field starting with column 73 (up to 4 characters).<br>These entries are interpreted only if option SOURCE or SOURCE and UPD are specified.<br>If input is from SYSDTA, the entries are ignored. |

| *COMOPT | Meaning |
|---|---|
| SOURCE= specification | Specifies the source from which the program is to be read.<br>If this option is omitted, the program is read from SYSDTA. For a full description of this option see Section 6.1.2. |
| SYSPARM= 'max. of 8 characters' | The system parameter &SYSPARM (an eight byte long character variable, see section 4.14.1.8) is assigned the specified entry and may be interpreted during macro processing. |
| UPD= specification | Specifies the update file from which corrections are to be read.<br>If this option is omitted, no corrections are expected or the corrections are initiated via SYSDTA by means of an *STARTC card (see Appendix A.8 and Section 6.1.2). |
| XREF | The cross reference listing is output. |
| NOXREF | Default assignment. No cross reference listing output. |

End of Option Input

| | Meaning |
|---|---|
| *END | End of *COMOPT statement input and start of assembly. Request for new options after assembly. |
| *END HALT | Same as *END, but termination of assembler after assembly. |
| *HALT | Immediate termination of assembler |

### 6.1.2 SOURCE Option and UPD Option

The SOURCE option may be used to specify the location of the source program, and the UPD option may be used to specify the location of the update file. If the SOURCE option is omitted, the source program will be read from SYSDTA. If the UPD option is omitted, the assumption is that there are no change lines.

```
                 /
                 +
specification:   *
                 file-name
                 lib(name)
                 plamlib(member[(version)])
```

If no entry is given for "specification", the source program will be read from SYSDTA.

/, + and * allowed for UPD.

/ — An interrupt will occur. SYSDTA can be assigned by the /SYSFILE command via SYSCMD. The source program or the change lines are then read in via SYSDTA. However, this reassignment of SYSDTA does not take effect until the options have been processed.

+ — The file specification of the source program or update file may be entered via the primary assignment of SYSDTA for this task (file from which the LOGON command was issued). The file specification may look as follows:

```
*
file-name
(SYSCMD)
(PRIMARY)
```

If *, (SYSCMD) or (PRIMARY) is specified, the source program or the change lines will be read from SYSDTA, in the latter case with reassignment of SYSDTA. Otherwise, reading in is effected via a file control block.

For UPD, only *, (SYSCMD) and (PRIMARY) are allowed.

* — The source program or the change lines are read from SYSDTA. The program text proper is preceded by options. The assembler does not, however, need the asterisk to recognize and process options preceding the actual text. Therefore, the asterisk entry in the source program input is equivalent to the missing entry of "specification".

file-name — Name of a cataloged file containing the source program or the change lines.
In this case, reading in is effected via a file control block. The name may be up to 54 characters in length, alphanumeric with period and hyphen.

| | |
|---|---|
| lib(name) | The source program is read from an LMS library.<br><br>lib is the file name of the library.<br>name is the name of the library member. |
| plamlib | File name of the program library created in accordance with LMS conventions, and from which the source program is to be read (member type = S).<br>In accordance with BS2000 conventions, the name may be up to 54 characters in length. |
| member | Name of the member of type = S, representing the source program.<br>The member name may be up to 62 characters in length. |
| version | Version number of the member.<br>version may be up to 3 characters in length.<br>If version is not specified, the member (type = S) with the highest available version number is used. |

Remarks

- The SOURCE option, in its entirety, must not exceed 80 characters, which implies that the assembler does not read any successor lines during option input.
  Hence, the maximum length of the expression in the SOURCE option to the right of the equal sign is 65 characters.

- Entries in the SOURCE option (library name, member name and version) are checked only for admissible length but not for correct syntax (similar to LMS conventions).

- If after *END any source program lines are expected from SYSDTA, a *STARTC card may follow first. The entries on the *STARTC card take priority. Thus a collision between *COMOPT UPD/SOURCE and the *STARTC card is avoided.

Attention

- The maximum length of the name of a called COPY member in the SOURCE section of a program library remains 8 characters.

- Tape processing is only supported via the *STARTC card.

### 6.1.3 MODULE Option

This option can be used to control output of the object module. If the option is omitted, the object module is output to the EAM file.

specification

```
{ *                                    }
{ plamlib[({*     }[(version)])]       }
{          {member}                    }
```

*  The object module is output to the EAM file.

plamlib  File name of the program library created in accordance with LMS conventions. The object module is stored as a member with type = R (module).
If no program library exists under the specified file name, one is created by the assembler.

member  Member name of the object module.
The member name should be in accordance with the "Rules for member designations in program libraries", cf LMS manual.
The member (type = R = module) is kept in the program library under this name (max. 8 characters). If * (asterisk) is specified, the member is given the name of the first program segment (CSECT name) of the object module. If the first program segment is unnamed, the member is stored <u>not</u> in the program library, but rather in the EAM file.
If the program does not contain a CSECT statement (or START statement), then the first DSECT or COM name is taken as the name of the object member. The same applies if only the library name is specified.

*

version  Version number of the member.
version may be up to 3 numerical characters in length.

<u>Character set supported by LMS</u>

Letters: A through Z
Digits: 0 through 9
Special characters: '.', '-', '@'

If version is not specified, the member is given the highest version number (represented in the program library by means of a '@' character, and in the terminal message by means of a '-' character).
If a member with the same version number already exists, this member is overwritten.

<u>Note</u>

The entries in the MODULE option are not syntax checked (see also the notes on the SOURCE option).

## 6.2 /PARAM COMMAND PARAMETERS

The PARAM command of the BS2000 command language is still evaluated, but *COMOPT statements take priority over identical operands in the /PARAM command.

The PARAM operands relevant to the assembler and the equivalent COMOPT options are shown in the following table.

| PARAM operand | COMOPT option |
|---|---|
| ASMLST=YES | LIST |
| ASMLST=NO | NOLIST |
| SAVLST=ALL | SAVLST |
| SAVLST=NO | NOSAVLST |
| SYMDIC=YES | ISD |
| SYMDIC=NO | NOISD |
| ERRFIL=YES | ERRFIL |
| ERRFIL=NO | NOERRFIL |
| XREF=YES | XREF |
| XREF=NO | NOXREF |
| ALTLIB=YES | ALTLIB |
| ALTLIB=NO | NOALTLIB |

| Process switch | COMOPT option |
|---|---|
| /SETSW ON=(0) | DUET |
| /SETSW OFF=(0) | NODUET |
| /SETSW ON=(31) | MCALL |
| /SETSW OFF=(31) | NOMCALL |

Note

As of Assembler Version 29.0, task switches should no longer be used

### 6.2.1 Description of PARAM operands

The following operands for the system /PARAM command may be used.

{A / B} A or B must be entered. If the operand is omitted, the underlined default value A is assumed.

**1. Cross-Reference Listing Option**

The following command indicates that a cross-reference listing is (or is not to be generated.

/PARAM XREF={YES / NO}

NO is the normal condition. If XREF = YES, the assembler places a combined symbol table map and cross-reference listing in the output file.

**2. Diagnostic File Option**

The following command indicates that a diagnostic file is (or is not) to be produced:

/PARAM SAVLST={ALL / NO}

If SAVLST=ALL, the assembler creates a file to be analyzed later by the Post-Assembly Diagnostic Routine (see section 2.7.6).

**3. Error File Option**

The following command indicates that an error file is (or is not) to be produced:

/PARAM ERRFIL={YES / NO}

If ERRFIL=YES, the assembler creates a reduced diagnostic file which lists invalid statements only (see section 2.7.6).

**4. Listing Option**

The following command indicates that an assembly listing is (or is not) to be created:

/PARAM ASMLST={YES / NO}

YES is the normal condition.
The assembler creates an assembly listing, if ASMLST=YES.

### 5. Internal Symbol Dictionary Option

The following command indicates that an Internal Symbol Dictionary is (or is not) to be produced:

/PARAM SYMDIC={YES / NO}

If SYMDIC=YES, the assembler produces ISD records which allow an error recovery within the symbols with the use of IDA.

### 6. TRANSDATA 960 (DUET) Statement Option

/SETSW ON=(0)

This command causes the assembler to recognize and assemble TRANSDATA 960 statements. It must be used when the assembler is to generate a DNSP (communication and network control program for TRANSDATA 960).

Note:

TASKSWITCH 0 may not be used by other programs.

### 7. Output of Object Modules to the EAM Object Module File

/PARAM CARD={YES / NO}

YES The object modules generated by the assembler are written to the EAM object module file of the task as well as to system file SYSOPT.

NO Object module output is suppressed unless DISC=YES is specified concurrently.

/PARAM DISC={YES / NO}

YES The object modules generated by the assembler are written to the EAM object module file of the task for further processing. The file name is "*".

NO Object module output is suppressed unless CARD=YES is specified.

## 6.3 ASSEMBLY AND LINKAGE EXAMPLES

Example 1 (dialog)

This sample job stream lists the commands to execute an assembly of a source program file on disc, to check the assembly with the aid of the Assembly Diagnostic Program and to do the linkage editing to the program.

```
/ERASE *
/EXEC $ASSEMB
**COMOPT SOURCE=TEST ———————————————————————— (1)
**COMOPT XREF,ISD,SAVLST ———————————————————— (2)
**END HALT

/EXEC $ADIAG
 OPEN SAVLST.ASSEMB.ICYL ———————————————————— (3)
 DISPLAY ——————————————————————————————————— (4)
 END

/EXEC $TSOSLNK
*PROG TESTA ——————————————————————————————— (5)
*INCLUDE *
*END

/EXEC TESTA ——————————————————————————————— (6)
```

(1) Name of the source file.

(2) Further assembly options.

(3) Open diagnostic file SAVLST.ASSEMB.ICYL (ICYL is the name of the first program section) and output assembly result in abbreviated form.

(4) Display causes of errors and numbers of the associated statements.

(5) Name of the program and name of the file to contain the linked program.

(6) Start program.

Example 2 (procedure)

Read source program from file SC.ASSEMB:

```
/SYSFILE SYSDTA=(SYSCMD)
/EXEC $ASSEMB
*COMOPT SOURCE=SC.ASSEMB
*END HALT
/
```

Example 3 (procedure)

Read source program from library PROGLIB, name of member PROG

```
/SYSFILE SYSDTA=(SYSCMD)
/EXEC $ASSEMB
*COMOPT SOURCE=PROGLIB(PROG)
*END HALT
/
```

Example 4

The assembly options are contained in the ASSPAR file.
The location of the source program should not be specified until the options have been processed, and the source program should be on file PROGM in a special case.

Contents of ASSPAR file

```
*COMOPT SOURCE=+
  .                 ┐
  .                 } further options
  .                 ┘
*END
```

Assembly

```
/SYSFILE SYSDTA=ASSPAR
/EXEC $ASSEMB
GIVE INPUT SPECIFICATION FOR SOURCE (assembler request)
PROGM
/
```

Alternative:
Contents of file ASSPAR

```
*COMOPT SOURCE=/
  .                 ┐
  .                 } further options
  .                 ┘
*END
```

Assembly

```
/SYSFILE SYSDTA=ASSPAR
/EXEC $ASSEMB
GIVE SYSFILE CMD TO READ SOURCE (assembler request)
/SYSFILE SYSDTA=PROGM
/R
```

## 6.4 ASSEMBLER TERMINAL AND ERROR MESSAGES

### 6.4.1 Normal Terminal Messages

In interactive mode, the following message appears on the terminal immediately after loading:

VERS.xxxx OF SIEMENS BS2000 ASSEMBLER READY
(only in interactive mode)

During the assembly (after the assembly listing has been output and before the XREF and/or diagnostic listings have been output) the following messages appear:

FLAGS IN xxxxx STATEMENTS, xxx PRIVILEGED FLAGS, xxx MNOTES

HIGHEST ERROR WEIGHT: {-, n} if no flag has been set. according to the highest error weight 0≤n≤2.

and, if used, the messages:

USER MACROLIBRARY: filename

2ND USER MACROLIBRARY: filename

3RD USER MACROLIBRARY: filename

4TH USER MACROLIBRARY: filename

5TH USER MACROLIBRARY: filename

SYSTEM MACROLIBRARY: filename

DIAGNOSTIC FILE: filename

ERRORFILE: filename

If the source program comes from an LMS library or from a library created in accordance with LMS conventions, the name of the library, the name of the member, its version number and its creation date are logged in the following format:

SOURCE LIBRARY: library (input library)
SOURCE PROGRAM: member (input member)
SOURCE VERS/DATE: ver/yymmdd

These messages are also output to SYSLST.

If the object module generated is output to a program library created in accordance with LMS conventions, the name of the library, the name of the member and the version number are logged in the following format:

```
MODULE LIBRARY:   library
LIBRARY ELEMENT:  member VERnnn
```

Upon assembly, the assembler displays the assembly time used:

```
ASSEMBLY TIME            x.xxx SEC.
```

This message is also output to SYSLST.

### 6.4.2 Messages on Reaching Break Condition

If the assembly is terminated prematurely, either when a fatal error (severity code 3) is encountered or when a MNOTE with error code 255 is generated, the message below appears on the terminal:

```
BREAK CONDITION REACHED ASSEMBLY TERMINATED
AT LEAST xxxx FLAGS RECOGNIZED - HIGHEST ERROR WEIGHT WAS n
```

### 6.4.3 Error Messages

If an internal assembler error is detected, a PDUMP and the following message is output on SYSLST only:

```
ERROR IN ROUTINE WAS CALLED, CHECK THE PDUMP ABOVE
```

The assembly is continued.

The following error messages are output to the printer and, in the interactive mode, also to the terminal.

**1. Macro Library Errors**

The error message depends on which macro library is currently in use:

```
ALTLIB[n]   text       (user library) (2 ≤ n ≤ 5)

SYSLIB      text       (system library)

text:       OPEN ERROR xxxx; ELEMENT: macro name
            READ ERROR xxxx; ELEMENT: macro name
            KEY ERROR; ELEMENT: macro name
```

xxxx is the error code supplied by the DMS (Data Management System).

After the message has been output the assembly is terminated with TERMJ.

Waiting time when opening a library:

```
{ALTLIB[n]
{SRCLIB    } LOCKED, NEXT ATTEMPT AFTER 6 SECONDS
{SYSLIB    }
```

This message is issued if the assembler attempts to open a library that has already been opened.
After 6 seconds another attempt is made to open the library. After 100 unsuccessful attempts, the following message is output:

```
{ALTLIB[n]
{SRCLIB    } OPEN ERROR 0D99
{SYSLIB    }
```

### 2. EAM Errors

After the following messages have been output, the assembly is terminated with TERMD.

Text:

EAM ERROR 01 ILLEGAL OPCODE [1]

EAM ERROR 02 ILLEGAL FILENAME

EAM ERROR 04 ILLEGAL BLOCKNUMBER [1]

EAM ERROR 08 ILLEGAL I/O ADDRESS [1]

EAM ERROR 10 EAM SPACE EXHAUSTED

EAM ERROR 20

EAM ERROR 40

EAM ERROR 80

EAM ERROR - KEYS DO NOT MATCH

Refer to the DMS Manual for a detailed description of the EAM error code.

[1] These error messages refer to assembler errors.

### 3. REQM and RELM Errors

After the error message has been output, the assembler run is terminated with TERMD.

Text:

REQM FAILED - INSUFFICIENT SPACE AVAILABLE - JOB TERMINATED

REQM FAILED - INVALID REQUEST - ASSEMB ERROR - JOB TERMINATED

RELM FAILED - JOB TERMINATED

**4. Error while generating a Diagnostic File**

If an error occurs while opening the file, this message is output:

OPEN ERROR xxxx(xxxx = DVS error code)

and a diagnostic file is not then generated.

If an error occurs while writing to the file, the following message appears:

ERROR WHILE WRITING TO DIAGNOSTIC FILE.

(no further action is taken).

The diagnostic file can accommodate a maximum of 98303 statements. If it is attempted to enter more than this number, the error message

DIAGNOSTIC FILE OVERFLOW - FILE CLOSED.

is output.



**5. Assembler Errors**

When "real" assembler errors occur, the assembly is terminated with ERFLG and TERMD, after the message has been output.

Text:

BAD CALL TO IOPACKAGE - JOB TERMINATED

ASSEMBLER ERROR - NO EOF SPECIFIED - JOB TERMINATED

ASSEMBF ERROR - BLOCKSIZE GREATER THAN 960 FOR OMF - JOB TERMINATED

INTERNAL ERROR TRYING TO READ BLOCK # GREATER THAN HIGHEST BLOCK NUMBER FOR THIS EAM FILE

**6. Errors while writing to SYSLST**

Text:

INTERNAL ERROR TRYING TO PRINT A LINE - IF
ND-LISTING IS REQUESTED, SYSLST MUST BE A
FILE - PROCESSING CONTINUES

**7. Errors while processing program libraries**

If internal errors are encountered while processing program libraries created in accordance with LMS conventions, one of the following messages is output and the assembly is terminated with TERMJ (see also next chapter: Warnings).

SOURCE LIBRARY NOT FOUND (OR WRONG LIB-TYPE)

This message is issued when an existing library cannot be opened.

SOURCE ELEMENT NOT FOUND
PLAM RETURNCODE error number

The meaning of the error numbers is described in the following manual: LMS (BS2000) Description (chapter 9).

**8. Miscellaneous Messages**

- When flags B44 or V43 occur in macro calls the following message is issued:

  WARNING: MACRO-CALL CAUSED B44 OR V43

The flags are output only if *COMOPT ERRPR=0.

- If a cataloged file with type=PAM is assigned for source input, the following message appears:

  ERROR: DISC-INPUT OPEN FAILED (SOURCE-INPUT = PAM-FILE)

- If after the assembler has been started module ICONTROL cannot be loaded, the following message is issued:

  ERROR: MODULE *ICONTROL* CANNOT BE LOADED ASSEMBLY TERMINATED

### 6.4.4 Warnings

**1. Warning when incompatible, non-privileged instructions are used.**

Whenever the instructions listed below are used, a 021-FLAG will be output.

In order to avoid compatibility problems the following should be borne in mind:

- Starting with BS2000 V8.0 the EXST instruction will no longer be supported and should therefore be removed from the program in due time.
- The LBF, STBF, LWI, STWI, PUSH and POP instructions should not be used in new programs. Their successful execution cannot be guaranteed in the long term.

**2. Warning when object module output to program libraries**

If internal errors are encountered when a program library is opened or when attaching to a program library, the assembly is not terminated, but rather the object module is output to the EAM file.
The warning below is issued in place of the normal terminal message:

MODULE LIBRARY: COULD NOT BE ATTACHED OR OPENED, MODULE IS PUT TO EAM
CAUSED BY PLAM RETURN CODE XXX

### 6.4.5 Assembler Error Flags

The assembler can diagnose up to 5 errors per statement. The errors are weighted and assigned one of four severity codes in accordance with their effect on execution of the generated object module:

| Severity code | Description | | |
|---|---|---|---|
| 0 | Warning | – | Successful program run possible |
| 1 | Error | – | Errored program run possible |
| 2 | Serious error | – | Successful program run unlikely |
| 3 | Fatal error | – | Assembly terminated |

The flag comprises a letter and a number (up to two digits).
If several errors are encountered in a single statement, then only a maximum of 3 letters for the flags are specified in the first few columns of the assembly listing.
A listing of all flags encountered, sorted according to letters, appears at the end of the assembly listing in the diagnostic listing. A brief description is appended and cross references are specified for each error.
If COMOPT FLGLST is used for the assembly, the error flags and descriptions are output after the invalid line. In such cases various errors are supplemented by additional information designating a symbol or specifying the approximate column of the error in the line. For technical reasons the column indicator is generally set to the end of an invalid character string.

Example:

```
A21     ***ERROR***NEGATIVE ADDRESS;NEAR OPERAND COLUMN 08
U10     ***ERROR***UNDEFINED SYMBOL;FELD
```

Note

Flags in conditional instructions or variable definitions within macros are only listed if the MTRAC statement is specified.

The list of error flags together with error identifiers and severity codes is contained in Appendix A.5.

### 6.4.6 Monitor Job Variable Support (MONJV)

#### 6.4.6.1 Function

The software product JV (Job Variables) permits jobs and programs running under BS2000 to be controlled and monitored (see "BS2000 Job Variables, Reference Manual"). The user defines a so-called monitoring job variable that he specifies as an operand of a LOGON, ENTER or EXEC command. The operating system enters in this job variable information about the current status of a program ("status indicator") as well as other information defined on program level ("return indicator"). The user can query this information at the end of the program; further jobs and programs may then be controlled dependent on this information.

After a program has been assembled, the monitoring job variable is supplied with a return code indicator.

Since multiple assemblies are allowed, the following applies:
The monitoring job variable contains the values from that assembly section where the highest error weight occurred.

#### 6.4.6.2 Assembler Indicators

**Return code**

The 4-digit return code stored in the MONJV has the following format:

TC PI

TC = Termination code, it may assume the following values:

C'0' Normal termination; no warnings or errors occurred.
C'1' Normal termination; warnings or class WAR/SIG/SER (see below) errors occurred.
C'2' Abnormal termination; an abort criterion has been reached set by the ERR=n/Sm option.
C'3' Abnormal termination; a class FAT or a compiler error has been identified.

PI = Program information; it may assume the following values:

C'000' No flags and no MNOTES reported (-).
C'001' Reserved
C'002' Highest error class 0, warnings, WAR.
C'003' Highest error class 1, significant error, SIG.
C'004' Highest error class 2, serious error, SER.
C'005' Highest error class 3, fatal error, FAT.
C'006' Compiler error

**Status indicator**

The 3-digit status indicator is set by the assembler as follows:

- Normal termination C'$T',
  at termination code C'0' or C'1'.

- Abnormal termination C'$A',
  at termination code C'2' or C'3'.

### 6.4.6.3 Potential combinations

| | | Ass. error MNOTE | – weight/ | Res. | WAR | SIG | SER | FAT | Comp. error |
|---|---|---|---|---|---|---|---|---|---|
| Status indi-cator | | PI | 000 | 001 | 002 | 003 | 004 | 005 | 006 |
| | TC | | | | | | | | |
| $T | 0 | | x | x | | | | | |
| $T | 1 | | | | x | x | x | | |
| $A | 2 *) | | | | x | x | x | | |
| $A | 3 *) | | | | | | | x | x |

*) In these cases a branch to a job step is made.

### 6.4.6.4 Interdependences

The MONJV will be supported starting with the assembler V29.1C.
The MONJV will not supported until Version 7.5 of BS2000. In versions < 7.5 the termination action of the assembler will remain unchanged.

## 6.5 LASER PRINTER ORIENTED ASSEMBLY LISTING

The laser printer oriented assembly listing (ND listing) is generated by specifying the NDLIST option in the *COMOPT statement. It differs from the standard listing on the following points:

1. All corrected or deleted source program cards are listed, together with the correction statements.

2. The ND listing is divided into three sections:

   Object code

   Source program

   Additional information

   Object code and source program are identical with the standard listing.

   Additional information consists of:

   - ISAM key, if the assembled program is contained in an ISAM file and SYSDTA was assigned.

   - Section names of symbols that represent addresses in instructions.

   - OPSYN listing lists the actual mnemonic operation code that was changed by means of an OPSYN statement.

   - STACK level gives the nesting level for each STACK or UNSTK statement.

     Ux für USING
     where $1 \leq x \leq 4$
     Px für PRINT

   - MTRAC information is output completely.

     Restriction: The value of SETC variables will be printed out up to a maximum of 50 characters.

3. In all cross reference and diagnostic listings the total of statement numbers will be increased from 13 to 24 per line.

   The following commands and statements are necessary:

```
/SYSFILE SYSLST=OUTFILE
/EXEC $ASSEMB
*COMOPT NDLIST,LINECNT=88,SOURCE=/
*END
/SYSFILE SYSDTA=INFILE
/R
/SYSFILE SYSLST=(PRIMARY)
/PRINT OUTFILE,LOOP=JI,FORM=code,CHARS=A7,SPACE=E
```

   INFILE — contains the source program

   OUTFILE — contains the ND listing after assembly

   code — is the computing center specific parameter that causes output to be sent to the laser printer.

   The ND listing is printed from the TSOS.NDFILE with the A7 character set and the VFB record: JI (see 3352 Laser Printer Reference Manual). With 88 lines per page, about 40 per cent of paper can be saved.

# A APPENDIX

## A.1 SUMMARY OF CONSTANTS

| Type | Implied Length (Bytes) | Alignment | Length Modifier Range | Specified by | Constants per Operand | Range for Exponents | Range for Scale | Truncation/ Padding Side |
|---|---|---|---|---|---|---|---|---|
| C | as needed | byte | 1 to 256 1) | characters | one | | | right |
| X | as needed | byte | 1 to 256 1) | hexadecimal digits | multiple | | | left |
| B | as needed | byte | 1 to 256 | binary digits | multiple | | | left |
| F | 4 | word | 1 to 8 | decimal digits | multiple | -85 to +75 | -187 to +346 | left |
| H | 2 | half word | 1 to 8 | decimal digits | multiple | -85 to +75 | -187 to +346 | left |
| E | 4 | word | 1 to 8 | decimal digits | multiple | -85 to +75 | 0 to 14 | right |
| D | 8 | double word | 1 to 8 | decimal digits | multiple | -85 to +75 | 0 to 14 | right |
| L | 16 | double word | 1 to 16 | decimal digits | multiple | -85 to +75 | 0 to 28 | right |
| P | as needed | byte | 1 to 16 | decimal digits | multiple | | | left |
| Z | as needed | byte | 1 to 16 | decimal digits | multiple | | | left |
| A | 4 | word | 1 to 4 | an absolute expression | multiple | | | left |
| | | | 3 or 4 | a relocatable or complex expression | | | | |
| V | 4 | word | 3 or 4 | relocatable symbol | multiple | | | left |

A

| Type | Implied Length (Bytes) | Alignment | Length Modifier Range | Specified by | Constants per Operand | Range for Exponents | Range for Scale | Truncation/ Padding Side |
|---|---|---|---|---|---|---|---|---|
| S | 2 | half word | 2 only | one absolute or relocatable expression or two absolute expressions: exp (exp) | multiple | | | |
| Y | 2 | half word | 1 or 2 | an absolute expression | multiple | | | left |
| | | | 2 only | a relocatable or complex relocatable expression | | | | |
| Q | 4 | word | 1 to 4 | symbolic address of an external dummy section | multiple | | | left |

(1) in a DS assembly instruction, C and X type constants may have length specifications up to 65,535 bytes.

<u>Note:</u> The duplication factor for DC constants is 65 535, for DS constants 16 777 215.

## A.2 RANGES OF MANTISSAS OF FLOATING-POINT CONSTANTS

| Type | Range of values (exact) | Range of values (approximately) |
|---|---|---|
| E | $16^{-65} \leq M \leq (1-16^{-6}) \times 16^{63}$ | $5{,}4 \times 10^{-79} \leq M \leq 7{,}2 \times 10^{75}$ |
| D | $16^{-65} \leq M \leq (1-16^{-14}) \times 16^{63}$ | |
| L | $16^{-65} \leq M \leq (1-16^{-28}) \times 16^{63}$ | |

If the constant is not within the valid range, it is flagged and not stored.

A

## A.3 CHARACTER CODES

| Decimal | Hexadecimal | EBCDIC | Character Set Punch Combination | Printer Graphics |
|---|---|---|---|---|
| 0 | 00 | 000 00000 | 12,0,9,8,1 | |
| 1 | 01 | 0000 0001 | 12,9,1 | |
| 2 | 02 | 0000 0010 | 12,9,2 | |
| 3 | 03 | 0000 0011 | 12,9,3 | |
| 4 | 04 | 0000 0100 | 12,9,4 | |
| 5 | 05 | 0000 0101 | 12,9,5 | |
| 6 | 06 | 0000 0110 | 12,9,6 | |
| 7 | 07 | 0000 0111 | 12,9,7 | |
| 8 | 08 | 0000 1000 | 12,9,8 | |
| 9 | 09 | 0000 1001 | 12,9,8,1 | |
| 10 | 0A | 0000 1010 | 12,9,8,2 | |
| 11 | 0B | 0000 1011 | 12,9,8,3 | |
| 12 | 0C | 0000 1100 | 12,9,8,4 | |
| 13 | 0D | 0000 1101 | 12,9,8,5 | |
| 14 | 0E | 0000 1110 | 12,9,8,6 | |
| 15 | 0F | 0000 1111 | 12,9,8,7 | |
| 16 | 10 | 0001 0000 | 12,11,9,8,1 | |
| 17 | 11 | 0001 0001 | 11,9,1 | |
| 18 | 12 | 0001 0010 | 11,9,2 | |
| 19 | 13 | 0001 0011 | 11,9,3 | |
| 20 | 14 | 0001 0100 | 11,9,4 | |
| 21 | 15 | 0001 0101 | 11,9,5 | |
| 22 | 16 | 0001 0110 | 11,9,6 | |
| 23 | 17 | 0001 0111 | 11,9,7 | |
| 24 | 18 | 0001 1000 | 11,9,8 | |
| 25 | 19 | 0001 1001 | 11,9,8,1 | |
| 26 | 1A | 0001 1010 | 11,9,8,2 | |
| 27 | 1B | 0001 1011 | 11,9,8,3 | |
| 28 | 1C | 0001 1100 | 11,9,8,4 | |
| 29 | 1D | 0001 1101 | 11,9,8,5 | |
| 30 | 1E | 0001 1110 | 11,9,8,6 | |
| 31 | 1F | 0001 1111 | 11,9,8,7 | |
| 32 | 20 | 0010 0000 | 11,0,9,8,1 | |
| 33 | 21 | 0010 0001 | 0,9,1 | |
| 34 | 22 | 0010 0010 | 0,9,2 | |
| 35 | 23 | 0010 0011 | 0,9,3 | |
| 36 | 24 | 0010 0100 | 0,9,4 | |
| 37 | 25 | 0010 0101 | 0,9,5 | |
| 38 | 26 | 0010 0110 | 0,9,6 | |
| 39 | 27 | 0010 0111 | 0,9,7 | |
| 40 | 28 | 0010 1000 | 0,9,8 | |
| 41 | 29 | 0010 1001 | 0,9,8,1 | |
| 42 | 2A | 0010 1010 | 0,9,8,2 | |
| 43 | 2B | 0010 1011 | 0,9,8,3 | |
| 44 | 2C | 0010 1100 | 0,9,8,4 | |
| 45 | 2D | 0010 1101 | 0,9,8,5 | |
| 46 | 2E | 0010 1110 | 0,9,8,6 | |
| 47 | 2F | 0010 1111 | 0,9,8,7 | |
| 48 | 30 | 0011 0000 | 12,11,0,9,8,1 | |
| 49 | 31 | 0011 0001 | 9,1 | |
| 50 | 32 | 0011 0010 | 9,2 | |
| 51 | 33 | 0011 0011 | 9,3 | |
| 52 | 34 | 0011 0100 | 9,4 | |
| 53 | 35 | 0011 0101 | 9,5 | |
| 54 | 36 | 0011 0110 | 9,6 | |

A

| Decimal | Hexadecimal | EBCDIC | Character Set Punch Combination | Printer Graphics |
|---|---|---|---|---|
| 55 | 37 | 0011 0111 | 9,7 | |
| 56 | 38 | 0011 1000 | 9,8 | |
| 57 | 39 | 0011 1001 | 9,8,1 | |
| 58 | 3A | 0011 1010 | 9,8,2 | |
| 59 | 3B | 0011 1011 | 9,8,3 | |
| 60 | 3C | 0011 1100 | 9,8,4 | |
| 61 | 3D | 0011 1101 | 9,8,5 | |
| 62 | 3E | 0011 1110 | 9,8,6 | |
| 63 | 3F | 0011 1111 | 9,8,7 | |
| 64 | 40 | 0100 0000 | | Space |
| 65 | 41 | 0100 0001 | 12,0,9,1 | |
| 66 | 42 | 0100 0010 | 12,0,9,2 | |
| 67 | 43 | 0100 0011 | 12,0,9,3 | |
| 68 | 44 | 0100 0100 | 12,0,9,4 | |
| 69 | 45 | 0100 0101 | 12,0,9,5 | |
| 70 | 46 | 0100 0110 | 12,0,9,6 | |
| 71 | 47 | 0100 0111 | 12,0,9,7 | |
| 72 | 48 | 0100 1000 | 12,0,9,8 | |
| 73 | 49 | 0100 1001 | 12,8,1 | |
| 74 | 4A | 0100 1010 | 12,8,2 | c (cents) |
| 75 | 4B | 0100 1011 | 12,8,3 | . (period) |
| 76 | 4C | 0100 1100 | 12,8,4 | < (less than) |
| 77 | 4D | 0100 1101 | 12,8,5 | ( (open parenthesis) |
| 78 | 4E | 0100 1110 | 12,8,6 | + (plus) |
| 79 | 4F | 0100 1111 | 12,8,7 | \| (vertical) |
| 80 | 50 | 0101 0000 | 12 | & (ampersand) |
| 81 | 51 | 0101 0001 | 12,11,9,1 | |
| 82 | 52 | 0101 0010 | 12,11,9,2 | |
| 83 | 53 | 0101 0011 | 12,11,9,3 | |
| 84 | 54 | 0101 0100 | 12,11,9,4 | |
| 85 | 55 | 0101 0101 | 12,11,9,5 | |
| 86 | 56 | 0101 0110 | 12,11,9,6 | |
| 87 | 57 | 0101 0111 | 12,11,9,7 | |
| 88 | 58 | 0101 1000 | 12,11,9,8 | |
| 89 | 59 | 0101 1001 | 11,8,1 | |
| 90 | 5A | 0101 1010 | 11,8,2 | ! (exclamation) |
| 91 | 5B | 0101 1011 | 11,8,3 | $ (dollar sign) |
| 92 | 5C | 0101 1100 | 11,8,4 | * (asterisk) |
| 93 | 5D | 0101 1101 | 11,8,5 | ) (close parenthesis) |
| 94 | 5E | 0101 1110 | 11,8,6 | ; (semicolon) |
| 95 | 5F | 0101 1111 | 11,8,7 | ¬ (logical NOT) |
| 96 | 60 | 0110 0000 | 11 | - (minus) |
| 97 | 61 | 0110 0001 | 0,1 | / (slash) |
| 98 | 62 | 0110 0010 | 11,0,9,2 | |
| 99 | 63 | 0110 0011 | 11,0,9,3 | |
| 100 | 64 | 0110 0100 | 11,0,9,4 | |
| 101 | 65 | 0110 0101 | 11,0,9,5 | |
| 102 | 66 | 0110 0110 | 11,0,9,6 | |
| 103 | 67 | 0110 0111 | 11,0,9,7 | |
| 104 | 68 | 0110 1000 | 11,0,9,8 | |
| 105 | 69 | 0110 1001 | 0,8,1 | |
| 106 | 6A | 0110 1010 | 12,11 | (logical AND) |
| 107 | 6B | 0110 1011 | 0,8,3 | , (comma) |
| 108 | 6C | 0110 1100 | 0,8,4 | % (percent) |
| 109 | 6D | 0110 1101 | 0,8,5 | _ (underline) |
| 110 | 6E | 0110 1110 | 0,8,6 | > (greater than) |
| 111 | 6F | 0110 1111 | 0,8,7 | ? (question mark) |
| 112 | 70 | 0111 0000 | 12,11,0 | |
| 113 | 71 | 0111 0001 | 12,11,0,9,1 | |

| Decimal | Hexadecimal | EBCDIC | Character Set Punch Combination | Printer Graphics |
|---|---|---|---|---|
| 114 | 72 | 0111 0010 | 12,11,0,9,2 | |
| 115 | 73 | 0111 0011 | 12,11,0,9,3 | |
| 116 | 74 | 0111 0100 | 12,11,0,9,4 | |
| 117 | 75 | 0111 0101 | 12,11,0,9,5 | |
| 118 | 76 | 0111 0110 | 12,11,0,9,6 | |
| 119 | 77 | 0111 0111 | 12,11,0,9,7 | |
| 120 | 78 | 0111 1000 | 12,11,0,9,8 | |
| 121 | 79 | 0111 1001 | 8,1 | |
| 122 | 7A | 0111 1010 | 8,2 | : (colon) |
| 123 | 7B | 0111 1011 | 8,3 | # (number sign) |
| 124 | 7C | 0111 1100 | 8,4 | @ (at the rate of) |
| 125 | 7D | 0111 1101 | 8,5 | ' (apostrophe) |
| 126 | 7E | 0111 1110 | 8,6 | = (equals) |
| 127 | 7F | 0111 1111 | 8,7 | " (quote) |
| 128 | 80 | 1000 0000 | 12,0,8,1 | |
| 129 | 81 | 1000 0001 | 12,0,1 | |
| 130 | 82 | 1000 0010 | 12,0,2 | |
| 131 | 83 | 1000 0011 | 12,0,3 | |
| 132 | 84 | 1000 0100 | 12,0,4 | |
| 133 | 85 | 1000 0101 | 12,0,5 | |
| 134 | 86 | 1000 0110 | 12,0,6 | |
| 135 | 87 | 1000 0111 | 12,0,7 | |
| 136 | 88 | 1000 1000 | 12,0,8 | |
| 137 | 89 | 1000 1001 | 12,0,9 | |
| 138 | 8A | 1000 1010 | 12,0,8,2 | |
| 139 | 8B | 1000 1011 | 12,0,8,3 | |
| 140 | 8C | 1000 1100 | 12,0,8,4 | |
| 141 | 8D | 1000 1101 | 12,0,8,5 | |
| 142 | 8E | 1000 1110 | 12,0,8,6 | |
| 143 | 8F | 1000 1111 | 12,0,8,7 | |
| 144 | 90 | 1001 0000 | 12,11,8,1 | |
| 145 | 91 | 1001 0001 | 12,11,1 | |
| 146 | 92 | 1001 0010 | 12,11,2 | |
| 147 | 93 | 1001 0011 | 12,11,3 | |
| 148 | 94 | 1001 0100 | 12,11,4 | |
| 149 | 95 | 1001 0101 | 12,11,5 | |
| 150 | 96 | 1001 0110 | 12,11,6 | |
| 151 | 97 | 1001 0111 | 12,11,7 | |
| 152 | 98 | 1001 1000 | 12,11,8 | |
| 153 | 99 | 1001 1001 | 12,11,9 | |
| 154 | 9A | 1001 1010 | 12,11,8,2 | |
| 155 | 9B | 1001 1011 | 12,11,8,3 | |
| 156 | 9C | 1001 1100 | 12,11,8,4 | |
| 157 | 9D | 1001 1101 | 12,11,8,5 | |
| 158 | 9E | 1001 1110 | 12,11,8,6 | |
| 159 | 9F | 1001 1111 | 12,11,8,7 | |
| 160 | A0 | 1010 0000 | 11,0,8,1 | |
| 161 | A1 | 1010 0001 | 11,0,1 | |
| 162 | A2 | 1010 0010 | 11,0,2 | |
| 163 | A3 | 1010 0011 | 11,0,3 | |
| 164 | A4 | 1010 0100 | 11,0,4 | |
| 165 | A5 | 1010 0101 | 11,0,5 | |
| 166 | A6 | 1010 0110 | 11,0,6 | |
| 167 | A7 | 1010 0111 | 11,0,7 | |
| 168 | A8 | 1010 1000 | 11,0,8 | |
| 169 | A9 | 1010 1001 | 11,0,9 | |
| 170 | AA | 1010 1010 | 11,0,8,2 | |
| 171 | AB | 1010 1011 | 11,0,8,3 | |
| 172 | AC | 1010 1100 | 11,0,8,4 | |

| Decimal | Hexadecimal | EBCDIC | Character Set Punch Combination | Printer Graphics |
|---|---|---|---|---|
| 173 | AD | 1010 1101 | 11,0,8,5 | |
| 174 | AE | 1010 1110 | 11,0,8,6 | |
| 175 | AF | 1010 1111 | 11,0,8,7 | |
| 176 | B0 | 1011 0000 | 12,11,0,8,1 | |
| 177 | B1 | 1011 0001 | 12,11,0,1 | |
| 178 | B2 | 1011 0010 | 12,11,0,2 | |
| 179 | B3 | 1011 0011 | 12,11,0,3 | |
| 180 | B4 | 1011 0100 | 12,11,0,4 | |
| 181 | B5 | 1011 0101 | 12,11,0,5 | |
| 182 | B6 | 1011 0110 | 12,11,0,6 | |
| 183 | B7 | 1011 0111 | 12,11,0,7 | |
| 184 | B8 | 1011 1000 | 12,11,0,8 | |
| 185 | B9 | 1011 1001 | 12,11,0,9 | |
| 186 | BA | 1011 1010 | 12,11,0,8,2 | |
| 187 | BB | 1011 1011 | 12,11,0,8,3 | |
| 188 | BC | 1011 1100 | 12,11,0,8,4 | |
| 189 | BD | 1011 1101 | 12,11,0,8,5 | |
| 190 | BE | 1011 1110 | 12,11,0,8,6 | |
| 191 | BF | 1011 1111 | 12,11,0,8,7 | |
| 192 | C0 | 1100 0000 | 12,0 | |
| 193 | C1 | 1100 0001 | 12,1 | A |
| 194 | C2 | 1100 0010 | 12,2 | B |
| 195 | C3 | 1100 0011 | 12,3 | C |
| 196 | C4 | 1100 0100 | 12,4 | D |
| 197 | C5 | 1100 0101 | 12,5 | E |
| 198 | C6 | 1100 0110 | 12,6 | F |
| 199 | C7 | 1100 0111 | 12,7 | G |
| 200 | C8 | 1100 1000 | 12,8 | H |
| 201 | C9 | 1100 1001 | 12,9 | I |
| 202 | CA | 1100 1010 | 12,0,9,8,2 | |
| 203 | CB | 1100 1011 | 12,0,9,8,3 | |
| 204 | CC | 1100 1100 | 12,0,9,8,4 | |
| 205 | CD | 1100 1101 | 12,0,9,8,5 | |
| 206 | CE | 1100 1110 | 12,0,9,8,6 | |
| 207 | CF | 1100 1111 | 12,0,9,8,7 | |
| 208 | D0 | 1101 0000 | 11,0 | |
| 209 | D1 | 1101 0001 | 11,1 | J |
| 210 | D2 | 1101 0010 | 11,2 | K |
| 211 | D3 | 1101 0011 | 11,3 | L |
| 212 | D4 | 1101 0100 | 11,4 | M |
| 213 | D5 | 1101 0101 | 11,5 | N |
| 214 | D6 | 1101 0110 | 11,6 | O |
| 215 | D7 | 1101 0111 | 11,7 | P |
| 216 | D8 | 1101 1000 | 11,8 | Q |
| 217 | D9 | 1101 1001 | 11,9 | R |
| 218 | DA | 1101 1010 | 12,11,9,8,2 | |
| 219 | DB | 1101 1011 | 12,11,9,8,3 | |
| 220 | DC | 1101 1100 | 12,11,9,8,4 | |
| 221 | DD | 1101 1101 | 12,11,9,8,5 | |
| 222 | DE | 1101 1110 | 12,11,9,8,6 | |
| 223 | DF | 1101 1111 | 12,11,9,8,7 | |
| 224 | E0 | 1110 0000 | 0,8,2 | |
| 225 | E1 | 1110 0001 | 11,0,9,1 | |
| 226 | E2 | 1110 0010 | 0,2 | S |
| 227 | E3 | 1110 0011 | 0,3 | T |
| 228 | E4 | 1110 0100 | 0,4 | U |
| 229 | E5 | 1110 0101 | 0,5 | V |
| 230 | E6 | 1110 0110 | 0,6 | W |

| Decimal | Hexadecimal | EBCDIC | Character Set Punch Combination | Printer Graphics |
|---|---|---|---|---|
| 231 | E7 | 1110 0111 | 0,7 | X |
| 232 | E8 | 1110 1000 | 0,8 | Y |
| 233 | E9 | 1110 1001 | 0,9 | Z |
| 234 | EA | 1110 1010 | 11,0,9,8,2 | |
| 235 | EB | 1110 1011 | 11,0,9,8,3 | |
| 236 | EC | 1110 1100 | 11,0,9,8,4 | |
| 237 | ED | 1110 1101 | 11,0,9,8,5 | |
| 238 | EE | 1110 1110 | 11,0,9,8,6 | |
| 239 | EF | 1110 1111 | 11,0,9,8,7 | |
| 240 | F0 | 1111 0000 | 0 | 0 |
| 241 | F1 | 1111 0001 | 1 | 1 |
| 242 | F2 | 1111 0010 | 2 | 2 |
| 243 | F3 | 1111 0011 | 3 | 3 |
| 244 | F4 | 1111 0100 | 4 | 4 |
| 245 | F5 | 1111 0101 | 5 | 5 |
| 246 | F6 | 1111 0110 | 6 | 6 |
| 247 | F7 | 1111 0111 | 7 | 7 |
| 248 | F8 | 1111 1000 | 8 | 8 |
| 249 | F9 | 1111 1001 | 9 | 9 |
| 250 | FA | 1111 1010 | 12,11,0,9,8,2 | |
| 251 | FB | 1111 1011 | 12,11,0,9,8,3 | |
| 252 | FC | 1111 1100 | 12,11,0,9,8,4 | |
| 253 | FD | 1111 1101 | 12,11,0,9,8,5 | |
| 254 | FE | 1111 1110 | 12,11,0,9,8,6 | |
| 255 | FF | 1111 1111 | 12,11,0,9,8,7 | (lozenge) |

A

# A.4 HEXADECIMAL-DECIMAL NUMBER CONVERSION

## 1. General

The table provides for direct conversion of hexadecimal and decimal numbers in these ranges:

| Hexadecimal | Decimal: |
|---|---|
| 000 to FFF | 0000 to 4095 |

## 2. Hexadecimal Decimal Number Conversion Table

In the table, the decimal value appears at the intersection of the row representing the most significant hexadecimal digits ($16^2$ and $16^1$) and the column representing the least significant hexadecimal digit ($16^0$).

Example: $C21_{16} = 3105_{10}$

| HEX | 0 | 1 | 2 |
|---|---|---|---|
| C0 | 3072 | 3073 | 3074 |
| C1 | 3088 | 3089 | 3090 |
| C2 | 3104 | 3105 | 3106 |
| C3 | 3120 | 3121 | 3122 |

For numbers outside the range of the table, add the following values to the table figures:

| Hexadecimal | Decimal | Hexadecimal | Decimal |
|---|---|---|---|
| 1000 | 4,096 | C000 | 49,152 |
| 2000 | 8,192 | D000 | 53,148 |
| 3000 | 12,288 | E000 | 57,344 |
| 4000 | 16,384 | F000 | 61,440 |
| 5000 | 20,480 | 10000 | 65,536 |
| 6000 | 24,576 | 20000 | 131,072 |
| 7000 | 28,672 | 30000 | 196,608 |
| 8000 | 32,768 | 40000 | 262,144 |
| 9000 | 36,864 | 50000 | 327,680 |
| A000 | 40,960 | 60000 | 393,216 |
| B000 | 45,056 | 70000 | 458,752 |

Example: $1C21_{16} = 7201_{10}$

| Hexadecimal | Decimal |
|---|---|
| C21 | 3105 |
| +1000 | +4096 |
| 1C21 | 7201 |

## Hexadecimal-Decimal number conversion

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 0000 | 0001 | 0002 | 0003 | 0004 | 0005 | 0006 | 0007 | 0008 | 0009 | 0010 | 0011 | 0012 | 0013 | 0014 | 0015 |
| 01 | 0016 | 0017 | 0018 | 0019 | 0020 | 0021 | 0022 | 0023 | 0024 | 0025 | 0026 | 0027 | 0028 | 0029 | 0030 | 0031 |
| 02 | 0032 | 0033 | 0034 | 0035 | 0036 | 0037 | 0038 | 0039 | 0040 | 0041 | 0042 | 0043 | 0044 | 0045 | 0046 | 0047 |
| 03 | 0048 | 0049 | 0050 | 0051 | 0052 | 0053 | 0054 | 0055 | 0056 | 0057 | 0058 | 0059 | 0060 | 0061 | 0062 | 0063 |
| 04 | 0064 | 0065 | 0066 | 0067 | 0068 | 0069 | 0070 | 0071 | 0072 | 0073 | 0074 | 0075 | 0076 | 0077 | 0078 | 0079 |
| 05 | 0080 | 0081 | 0082 | 0083 | 0084 | 0085 | 0086 | 0087 | 0088 | 0089 | 0090 | 0091 | 0092 | 0093 | 0094 | 0095 |
| 06 | 0096 | 0097 | 0098 | 0099 | 0100 | 0101 | 0102 | 0103 | 0104 | 0105 | 0106 | 0107 | 0108 | 0109 | 0110 | 0111 |
| 07 | 0112 | 0113 | 0114 | 0115 | 0116 | 0117 | 0118 | 0119 | 0120 | 0121 | 0122 | 0123 | 0124 | 0125 | 0126 | 0127 |
| 08 | 0128 | 0129 | 0130 | 0131 | 0132 | 0133 | 0134 | 0135 | 0136 | 0137 | 0138 | 0139 | 0140 | 0141 | 0142 | 0143 |
| 09 | 0144 | 0145 | 0146 | 0147 | 0148 | 0149 | 0150 | 0151 | 0152 | 0153 | 0154 | 0155 | 0156 | 0157 | 0158 | 0159 |
| 0A | 0160 | 0161 | 0162 | 0163 | 0164 | 0165 | 0166 | 0167 | 0168 | 0169 | 0170 | 0171 | 0172 | 0173 | 0174 | 0175 |
| 0B | 0176 | 0177 | 0178 | 0179 | 0180 | 0181 | 0182 | 0183 | 0184 | 0185 | 0186 | 0187 | 0188 | 0189 | 0190 | 0191 |
| 0C | 0192 | 0193 | 0194 | 0195 | 0196 | 0197 | 0198 | 0199 | 0200 | 0201 | 0202 | 0203 | 0204 | 0205 | 0206 | 0207 |
| 0D | 0208 | 0209 | 0210 | 0211 | 0212 | 0213 | 0214 | 0215 | 0216 | 0217 | 0218 | 0219 | 0220 | 0221 | 0222 | 0223 |
| 0E | 0224 | 0225 | 0226 | 0227 | 0228 | 0229 | 0230 | 0231 | 0232 | 0233 | 0234 | 0235 | 0236 | 0237 | 0238 | 0239 |
| 0F | 0240 | 0241 | 0242 | 0243 | 0244 | 0245 | 0246 | 0247 | 0248 | 0249 | 0250 | 0251 | 0252 | 0253 | 0254 | 0255 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 0256 | 0257 | 0258 | 0259 | 0260 | 0261 | 0262 | 0263 | 0264 | 0265 | 0266 | 0267 | 0268 | 0269 | 0270 | 0271 |
| 11 | 0272 | 0273 | 0274 | 0275 | 0276 | 0277 | 0278 | 0279 | 0280 | 0281 | 0282 | 0283 | 0284 | 0285 | 0286 | 0287 |
| 12 | 0288 | 0289 | 0290 | 0291 | 0292 | 0293 | 0294 | 0295 | 0296 | 0297 | 0298 | 0299 | 0300 | 0301 | 0302 | 0303 |
| 13 | 0304 | 0305 | 0306 | 0307 | 0308 | 0309 | 0310 | 0311 | 0312 | 0313 | 0314 | 0315 | 0316 | 0317 | 0318 | 0319 |
| 14 | 0320 | 0321 | 0322 | 0323 | 0324 | 0325 | 0326 | 0327 | 0328 | 0329 | 0330 | 0331 | 0332 | 0333 | 0334 | 0335 |
| 15 | 0336 | 0337 | 0338 | 0339 | 0340 | 0341 | 0342 | 0343 | 0344 | 0345 | 0346 | 0347 | 0348 | 0349 | 0350 | 0351 |
| 16 | 0352 | 0353 | 0354 | 0355 | 0356 | 0357 | 0358 | 0359 | 0360 | 0361 | 0362 | 0363 | 0364 | 0365 | 0366 | 0367 |
| 17 | 0368 | 0369 | 0370 | 0371 | 0372 | 0373 | 0374 | 0375 | 0376 | 0377 | 0378 | 0379 | 0380 | 0381 | 0382 | 0383 |
| 18 | 0384 | 0385 | 0386 | 0387 | 0388 | 0389 | 0390 | 0391 | 0392 | 0393 | 0394 | 0395 | 0396 | 0397 | 0398 | 0399 |
| 19 | 0400 | 0401 | 0402 | 0403 | 0404 | 0405 | 0406 | 0407 | 0408 | 0409 | 0410 | 0411 | 0412 | 0413 | 0414 | 0415 |
| 1A | 0416 | 0417 | 0418 | 0419 | 0420 | 0421 | 0422 | 0423 | 0424 | 0425 | 0426 | 0427 | 0428 | 0429 | 0430 | 0431 |
| 1B | 0432 | 0433 | 0434 | 0435 | 0436 | 0437 | 0438 | 0439 | 0440 | 0441 | 0442 | 0443 | 0444 | 0445 | 0446 | 0447 |
| 1C | 0448 | 0449 | 0450 | 0451 | 0452 | 0453 | 0454 | 0455 | 0456 | 0457 | 0458 | 0459 | 0460 | 0461 | 0462 | 0463 |
| 1D | 0464 | 0465 | 0466 | 0467 | 0468 | 0469 | 0470 | 0471 | 0472 | 0473 | 0474 | 0475 | 0476 | 0477 | 0478 | 0479 |
| 1E | 0480 | 0481 | 0482 | 0483 | 0484 | 0485 | 0486 | 0487 | 0488 | 0489 | 0490 | 0491 | 0492 | 0493 | 0494 | 0495 |
| 1F | 0496 | 0497 | 0498 | 0499 | 0500 | 0501 | 0502 | 0503 | 0504 | 0505 | 0506 | 0507 | 0508 | 0509 | 0510 | 0511 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 0512 | 0513 | 0514 | 0515 | 0516 | 0517 | 0518 | 0519 | 0520 | 0521 | 0522 | 0523 | 0524 | 0525 | 0526 | 0527 |
| 21 | 0528 | 0529 | 0530 | 0531 | 0532 | 0533 | 0534 | 0535 | 0536 | 0537 | 0538 | 0539 | 0540 | 0541 | 0542 | 0543 |
| 22 | 0544 | 0545 | 0546 | 0547 | 0548 | 0549 | 0550 | 0551 | 0552 | 0553 | 0554 | 0555 | 0556 | 0557 | 0558 | 0559 |
| 23 | 0560 | 0561 | 0562 | 0563 | 0564 | 0565 | 0566 | 0567 | 0568 | 0569 | 0570 | 0571 | 0572 | 0573 | 0574 | 0575 |
| 24 | 0576 | 0577 | 0578 | 0579 | 0580 | 0581 | 0582 | 0583 | 0584 | 0585 | 0586 | 0587 | 0588 | 0589 | 0590 | 0591 |
| 25 | 0592 | 0593 | 0594 | 0595 | 0596 | 0597 | 0598 | 0599 | 0600 | 0601 | 0602 | 0603 | 0604 | 0605 | 0606 | 0607 |
| 26 | 0608 | 0609 | 0610 | 0611 | 0612 | 0613 | 0614 | 0615 | 0616 | 0617 | 0618 | 0619 | 0620 | 0621 | 0622 | 0623 |
| 27 | 0624 | 0625 | 0626 | 0627 | 0628 | 0629 | 0630 | 0631 | 0632 | 0633 | 0634 | 0635 | 0636 | 0637 | 0638 | 0639 |
| 28 | 0640 | 0641 | 0642 | 0643 | 0644 | 0645 | 0646 | 0647 | 0648 | 0649 | 0650 | 0651 | 0652 | 0653 | 0654 | 0655 |
| 29 | 0656 | 0657 | 0658 | 0659 | 0660 | 0661 | 0662 | 0663 | 0664 | 0665 | 0666 | 0667 | 0668 | 0669 | 0670 | 0671 |
| 2A | 0672 | 0673 | 0674 | 0675 | 0676 | 0677 | 0678 | 0679 | 0680 | 0681 | 0682 | 0683 | 0684 | 0685 | 0686 | 0687 |
| 2B | 0688 | 0689 | 0690 | 0691 | 0692 | 0693 | 0694 | 0695 | 0696 | 0697 | 0698 | 0699 | 0700 | 0701 | 0702 | 0703 |
| 2C | 0704 | 0705 | 0706 | 0707 | 0708 | 0709 | 0710 | 0711 | 0712 | 0713 | 0714 | 0715 | 0716 | 0717 | 0718 | 0719 |
| 2D | 0720 | 0721 | 0722 | 0723 | 0724 | 0725 | 0726 | 0727 | 0728 | 0729 | 0730 | 0731 | 0732 | 0733 | 0734 | 0735 |
| 2E | 0736 | 0737 | 0738 | 0739 | 0740 | 0741 | 0742 | 0743 | 0744 | 0745 | 0746 | 0747 | 0748 | 0749 | 0750 | 0751 |
| 2F | 0752 | 0753 | 0754 | 0755 | 0756 | 0757 | 0758 | 0759 | 0760 | 0761 | 0762 | 0763 | 0764 | 0765 | 0766 | 0767 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 30 | 0768 | 0769 | 0770 | 0771 | 0772 | 0773 | 0774 | 0775 | 0776 | 0777 | 0778 | 0779 | 0780 | 0781 | 0782 | 0783 |
| 31 | 0784 | 0785 | 0786 | 0787 | 0788 | 0789 | 0790 | 0791 | 0792 | 0793 | 0794 | 0795 | 0796 | 0797 | 0798 | 0799 |
| 32 | 0800 | 0801 | 0802 | 0803 | 0804 | 0805 | 0806 | 0807 | 0808 | 0809 | 0810 | 0811 | 0812 | 0813 | 0814 | 0815 |
| 33 | 0816 | 0817 | 0818 | 0819 | 0820 | 0821 | 0822 | 0823 | 0824 | 0825 | 0826 | 0827 | 0828 | 0829 | 0830 | 0831 |
| 34 | 0832 | 0833 | 0834 | 0835 | 0836 | 0837 | 0838 | 0839 | 0840 | 0841 | 0842 | 0843 | 0844 | 0845 | 0846 | 0847 |
| 35 | 0848 | 0849 | 0850 | 0851 | 0852 | 0853 | 0854 | 0855 | 0856 | 0857 | 0858 | 0859 | 0860 | 0861 | 0862 | 0863 |
| 36 | 0864 | 0865 | 0866 | 0867 | 0868 | 0869 | 0870 | 0871 | 0872 | 0873 | 0874 | 0875 | 0876 | 0877 | 0878 | 0879 |
| 37 | 0880 | 0881 | 0882 | 0883 | 0884 | 0885 | 0886 | 0887 | 0888 | 0889 | 0890 | 0891 | 0892 | 0893 | 0894 | 0895 |
| 38 | 0896 | 0897 | 0898 | 0899 | 0900 | 0901 | 0902 | 0903 | 0904 | 0905 | 0906 | 0907 | 0908 | 0909 | 0910 | 0911 |
| 39 | 0912 | 0913 | 0914 | 0915 | 0916 | 0917 | 0918 | 0919 | 0920 | 0921 | 0922 | 0923 | 0924 | 0925 | 0926 | 0927 |
| 3A | 0928 | 0929 | 0930 | 0931 | 0932 | 0933 | 0934 | 0935 | 0936 | 0937 | 0938 | 0939 | 0940 | 0941 | 0942 | 0943 |
| 3B | 0944 | 0945 | 0946 | 0947 | 0948 | 0949 | 0950 | 0951 | 0952 | 0953 | 0954 | 0955 | 0956 | 0957 | 0958 | 0959 |
| 3C | 0960 | 0961 | 0962 | 0963 | 0964 | 0965 | 0966 | 0967 | 0968 | 0969 | 0970 | 0971 | 0972 | 0973 | 0974 | 0975 |
| 3D | 0976 | 0977 | 0978 | 0979 | 0980 | 0981 | 0982 | 0983 | 0984 | 0985 | 0986 | 0987 | 0988 | 0989 | 0990 | 0991 |
| 3E | 0992 | 0993 | 0994 | 0995 | 0996 | 0997 | 0998 | 0999 | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 |
| 3F | 1008 | 1009 | 1010 | 1011 | 1012 | 1013 | 1014 | 1015 | 1016 | 1017 | 1018 | 1019 | 1020 | 1021 | 1022 | 1023 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 40 | 1024 | 1025 | 1026 | 1027 | 1028 | 1029 | 1030 | 1031 | 1032 | 1033 | 1034 | 1035 | 1036 | 1037 | 1038 | 1039 |
| 41 | 1040 | 1041 | 1042 | 1043 | 1044 | 1045 | 1046 | 1047 | 1048 | 1049 | 1050 | 1051 | 1052 | 1053 | 1054 | 1055 |
| 42 | 1056 | 1057 | 1058 | 1059 | 1060 | 1061 | 1062 | 1063 | 1064 | 1065 | 1066 | 1067 | 1068 | 1069 | 1070 | 1071 |
| 43 | 1072 | 1073 | 1074 | 1075 | 1076 | 1077 | 1078 | 1079 | 1080 | 1081 | 1082 | 1083 | 1084 | 1085 | 1086 | 1087 |
| 44 | 1088 | 1089 | 1090 | 1091 | 1092 | 1093 | 1094 | 1095 | 1096 | 1097 | 1098 | 1099 | 1100 | 1101 | 1102 | 1103 |
| 45 | 1104 | 1105 | 1106 | 1107 | 1108 | 1109 | 1110 | 1111 | 1112 | 1113 | 1114 | 1115 | 1116 | 1117 | 1118 | 1119 |
| 46 | 1120 | 1121 | 1122 | 1123 | 1124 | 1125 | 1126 | 1127 | 1128 | 1129 | 1130 | 1131 | 1132 | 1133 | 1134 | 1135 |
| 47 | 1136 | 1137 | 1138 | 1139 | 1140 | 1141 | 1142 | 1143 | 1144 | 1145 | 1146 | 1147 | 1148 | 1149 | 1150 | 1151 |
| 48 | 1152 | 1153 | 1154 | 1155 | 1156 | 1157 | 1158 | 1159 | 1160 | 1161 | 1162 | 1163 | 1164 | 1165 | 1166 | 1167 |
| 49 | 1168 | 1169 | 1170 | 1171 | 1172 | 1173 | 1174 | 1175 | 1176 | 1177 | 1178 | 1179 | 1180 | 1181 | 1182 | 1183 |
| 4A | 1184 | 1185 | 1186 | 1187 | 1188 | 1189 | 1190 | 1191 | 1192 | 1193 | 1194 | 1195 | 1196 | 1197 | 1198 | 1199 |
| 4B | 1200 | 1201 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 | 1208 | 1209 | 1210 | 1211 | 1212 | 1213 | 1214 | 1215 |
| 4C | 1216 | 1217 | 1218 | 1219 | 1220 | 1221 | 1222 | 1223 | 1224 | 1225 | 1226 | 1227 | 1228 | 1229 | 1230 | 1231 |
| 4D | 1232 | 1233 | 1234 | 1235 | 1236 | 1237 | 1238 | 1239 | 1240 | 1241 | 1242 | 1243 | 1244 | 1245 | 1246 | 1247 |
| 4E | 1248 | 1249 | 1250 | 1251 | 1252 | 1253 | 1254 | 1255 | 1256 | 1257 | 1258 | 1259 | 1260 | 1261 | 1262 | 1263 |
| 4F | 1264 | 1265 | 1266 | 1267 | 1268 | 1269 | 1270 | 1271 | 1272 | 1273 | 1274 | 1275 | 1276 | 1277 | 1278 | 1279 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 50 | 1280 | 1281 | 1282 | 1283 | 1284 | 1285 | 1286 | 1287 | 1288 | 1289 | 1290 | 1291 | 1292 | 1293 | 1294 | 1295 |
| 51 | 1296 | 1297 | 1298 | 1299 | 1300 | 1301 | 1302 | 1303 | 1304 | 1305 | 1306 | 1307 | 1308 | 1309 | 1310 | 1311 |
| 52 | 1312 | 1313 | 1314 | 1315 | 1316 | 1317 | 1318 | 1319 | 1320 | 1321 | 1322 | 1323 | 1324 | 1325 | 1326 | 1327 |
| 53 | 1328 | 1329 | 1330 | 1331 | 1332 | 1333 | 1334 | 1335 | 1336 | 1337 | 1338 | 1339 | 1340 | 1341 | 1342 | 1343 |
| 54 | 1344 | 1345 | 1346 | 1347 | 1348 | 1349 | 1350 | 1351 | 1352 | 1353 | 1354 | 1355 | 1356 | 1357 | 1358 | 1359 |
| 55 | 1360 | 1361 | 1362 | 1363 | 1364 | 1365 | 1366 | 1367 | 1368 | 1369 | 1370 | 1371 | 1372 | 1373 | 1374 | 1375 |
| 56 | 1376 | 1377 | 1378 | 1379 | 1380 | 1381 | 1382 | 1383 | 1384 | 1385 | 1386 | 1387 | 1388 | 1389 | 1390 | 1391 |
| 57 | 1392 | 1393 | 1394 | 1395 | 1396 | 1397 | 1398 | 1399 | 1400 | 1401 | 1402 | 1403 | 1404 | 1405 | 1406 | 1407 |
| 58 | 1408 | 1409 | 1410 | 1411 | 1412 | 1413 | 1414 | 1415 | 1416 | 1417 | 1418 | 1419 | 1420 | 1421 | 1422 | 1423 |
| 59 | 1424 | 1425 | 1426 | 1427 | 1428 | 1429 | 1430 | 1431 | 1432 | 1433 | 1434 | 1435 | 1436 | 1437 | 1438 | 1439 |
| 5A | 1440 | 1441 | 1442 | 1443 | 1444 | 1445 | 1446 | 1447 | 1448 | 1449 | 1450 | 1451 | 1452 | 1453 | 1454 | 1455 |
| 5B | 1456 | 1457 | 1458 | 1459 | 1460 | 1461 | 1462 | 1463 | 1464 | 1465 | 1466 | 1467 | 1468 | 1469 | 1470 | 1471 |
| 5C | 1472 | 1473 | 1474 | 1475 | 1476 | 1477 | 1478 | 1479 | 1480 | 1481 | 1482 | 1483 | 1484 | 1485 | 1486 | 1487 |
| 5D | 1488 | 1489 | 1490 | 1491 | 1492 | 1493 | 1494 | 1495 | 1496 | 1497 | 1498 | 1499 | 1500 | 1501 | 1502 | 1503 |
| 5E | 1504 | 1505 | 1506 | 1507 | 1508 | 1509 | 1510 | 1511 | 1512 | 1513 | 1514 | 1515 | 1516 | 1517 | 1518 | 1519 |
| 5F | 1520 | 1521 | 1522 | 1523 | 1524 | 1525 | 1526 | 1527 | 1528 | 1529 | 1530 | 1531 | 1532 | 1533 | 1534 | 1535 |

A

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 60 | 1536 | 1537 | 1538 | 1539 | 1540 | 1541 | 1542 | 1543 | 1544 | 1545 | 1546 | 1547 | 1548 | 1549 | 1550 | 1551 |
| 61 | 1552 | 1553 | 1554 | 1555 | 1556 | 1557 | 1558 | 1559 | 1560 | 1561 | 1562 | 1563 | 1564 | 1565 | 1566 | 1567 |
| 62 | 1568 | 1569 | 1570 | 1571 | 1572 | 1573 | 1574 | 1575 | 1576 | 1577 | 1578 | 1579 | 1580 | 1581 | 1582 | 1583 |
| 63 | 1584 | 1585 | 1586 | 1587 | 1588 | 1589 | 1590 | 1591 | 1592 | 1593 | 1594 | 1595 | 1596 | 1597 | 1598 | 1599 |
| 64 | 1600 | 1601 | 1602 | 1603 | 1604 | 1605 | 1606 | 1607 | 1608 | 1609 | 1610 | 1611 | 1612 | 1613 | 1614 | 1615 |
| 65 | 1616 | 1617 | 1618 | 1619 | 1620 | 1621 | 1622 | 1623 | 1624 | 1625 | 1626 | 1627 | 1628 | 1629 | 1630 | 1631 |
| 66 | 1632 | 1633 | 1634 | 1635 | 1636 | 1637 | 1638 | 1639 | 1640 | 1641 | 1642 | 1643 | 1644 | 1645 | 1646 | 1647 |
| 67 | 1648 | 1649 | 1650 | 1651 | 1652 | 1653 | 1654 | 1655 | 1656 | 1657 | 1658 | 1659 | 1660 | 1661 | 1662 | 1663 |
| 68 | 1664 | 1665 | 1666 | 1667 | 1668 | 1669 | 1670 | 1671 | 1672 | 1673 | 1674 | 1675 | 1676 | 1677 | 1678 | 1679 |
| 69 | 1680 | 1681 | 1682 | 1683 | 1684 | 1685 | 1686 | 1687 | 1688 | 1689 | 1690 | 1691 | 1692 | 1693 | 1694 | 1695 |
| 6A | 1696 | 1697 | 1698 | 1699 | 1700 | 1701 | 1702 | 1703 | 1704 | 1705 | 1706 | 1707 | 1708 | 1709 | 1710 | 1711 |
| 6B | 1712 | 1713 | 1714 | 1715 | 1716 | 1717 | 1718 | 1719 | 1720 | 1721 | 1722 | 1723 | 1724 | 1725 | 1726 | 1727 |
| 6C | 1728 | 1729 | 1730 | 1731 | 1732 | 1733 | 1734 | 1735 | 1736 | 1737 | 1738 | 1739 | 1740 | 1741 | 1742 | 1743 |
| 6D | 1744 | 1745 | 1746 | 1747 | 1748 | 1749 | 1750 | 1751 | 1752 | 1753 | 1754 | 1755 | 1756 | 1757 | 1758 | 1759 |
| 6E | 1760 | 1761 | 1762 | 1763 | 1764 | 1765 | 1766 | 1767 | 1768 | 1769 | 1770 | 1771 | 1772 | 1773 | 1774 | 1775 |
| 6F | 1776 | 1777 | 1778 | 1779 | 1780 | 1781 | 1782 | 1783 | 1784 | 1785 | 1786 | 1787 | 1788 | 1789 | 1790 | 1791 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 70 | 1792 | 1793 | 1794 | 1795 | 1796 | 1797 | 1798 | 1799 | 1800 | 1801 | 1802 | 1803 | 1804 | 1805 | 1806 | 1807 |
| 71 | 1808 | 1809 | 1810 | 1811 | 1812 | 1813 | 1814 | 1815 | 1816 | 1817 | 1818 | 1819 | 1820 | 1821 | 1822 | 1823 |
| 72 | 1824 | 1825 | 1826 | 1827 | 1828 | 1829 | 1830 | 1831 | 1832 | 1833 | 1834 | 1835 | 1836 | 1837 | 1838 | 1839 |
| 73 | 1840 | 1841 | 1842 | 1843 | 1844 | 1845 | 1846 | 1847 | 1848 | 1849 | 1850 | 1851 | 1852 | 1853 | 1854 | 1855 |
| 74 | 1856 | 1857 | 1858 | 1859 | 1860 | 1861 | 1862 | 1863 | 1864 | 1865 | 1866 | 1867 | 1868 | 1869 | 1870 | 1871 |
| 75 | 1872 | 1873 | 1874 | 1875 | 1876 | 1877 | 1878 | 1879 | 1880 | 1881 | 1882 | 1883 | 1884 | 1885 | 1886 | 1887 |
| 76 | 1888 | 1889 | 1890 | 1891 | 1892 | 1893 | 1894 | 1895 | 1896 | 1897 | 1898 | 1899 | 1900 | 1901 | 1902 | 1903 |
| 77 | 1904 | 1905 | 1906 | 1907 | 1908 | 1909 | 1910 | 1911 | 1912 | 1913 | 1914 | 1915 | 1916 | 1917 | 1918 | 1919 |
| 78 | 1920 | 1921 | 1922 | 1923 | 1924 | 1925 | 1926 | 1927 | 1928 | 1929 | 1930 | 1931 | 1932 | 1933 | 1934 | 1935 |
| 79 | 1936 | 1937 | 1938 | 1939 | 1940 | 1941 | 1942 | 1943 | 1944 | 1945 | 1946 | 1947 | 1948 | 1949 | 1950 | 1951 |
| 7A | 1952 | 1953 | 1954 | 1955 | 1956 | 1957 | 1958 | 1959 | 1960 | 1961 | 1962 | 1963 | 1964 | 1965 | 1966 | 1967 |
| 7B | 1968 | 1969 | 1970 | 1971 | 1972 | 1973 | 1974 | 1975 | 1976 | 1977 | 1978 | 1979 | 1980 | 1981 | 1982 | 1983 |
| 7C | 1984 | 1985 | 1986 | 1987 | 1988 | 1989 | 1990 | 1991 | 1992 | 1993 | 1994 | 1995 | 1996 | 1997 | 1998 | 1999 |
| 7D | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 |
| 7E | 2016 | 2017 | 2018 | 2019 | 2020 | 2021 | 2022 | 2023 | 2024 | 2025 | 2026 | 2027 | 2028 | 2029 | 2030 | 2031 |
| 7F | 2032 | 2033 | 2034 | 2035 | 2036 | 2037 | 2038 | 2039 | 2040 | 2041 | 2042 | 2043 | 2044 | 2045 | 2046 | 2047 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80 | 2048 | 2049 | 2050 | 2051 | 2052 | 2053 | 2054 | 2055 | 2056 | 2057 | 2058 | 2059 | 2060 | 2061 | 2062 | 2063 |
| 81 | 2064 | 2065 | 2066 | 2067 | 2068 | 2069 | 2070 | 2071 | 2072 | 2073 | 2074 | 2075 | 2076 | 2077 | 2078 | 2079 |
| 82 | 2080 | 2081 | 2082 | 2083 | 2084 | 2085 | 2086 | 2087 | 2088 | 2089 | 2090 | 2091 | 2092 | 2093 | 2094 | 2095 |
| 83 | 2096 | 2097 | 2098 | 2099 | 2100 | 2101 | 2102 | 2103 | 2104 | 2105 | 2106 | 2107 | 2108 | 2109 | 2110 | 2111 |
| 84 | 2112 | 2113 | 2114 | 2115 | 2116 | 2117 | 2118 | 2119 | 2120 | 2121 | 2122 | 2123 | 2124 | 2125 | 2126 | 2127 |
| 85 | 2128 | 2129 | 2130 | 2131 | 2132 | 2133 | 2134 | 2135 | 2136 | 2137 | 2138 | 2139 | 2140 | 2141 | 2142 | 2143 |
| 86 | 2144 | 2145 | 2146 | 2147 | 2148 | 2149 | 2150 | 2151 | 2152 | 2153 | 2154 | 2155 | 2156 | 2157 | 2158 | 2159 |
| 87 | 2160 | 2161 | 2162 | 2163 | 2164 | 2165 | 2166 | 2167 | 2168 | 2169 | 2170 | 2171 | 2172 | 2173 | 2174 | 2175 |
| 88 | 2176 | 2177 | 2178 | 2179 | 2180 | 2181 | 2182 | 2183 | 2184 | 2185 | 2186 | 2187 | 2188 | 2189 | 2190 | 2191 |
| 89 | 2192 | 2193 | 2194 | 2195 | 2196 | 2197 | 2198 | 2199 | 2200 | 2201 | 2202 | 2203 | 2204 | 2205 | 2206 | 2207 |
| 8A | 2208 | 2209 | 2210 | 2211 | 2212 | 2213 | 2214 | 2215 | 2216 | 2217 | 2218 | 2219 | 2220 | 2221 | 2222 | 2223 |
| 8B | 2224 | 2225 | 2226 | 2227 | 2228 | 2229 | 2230 | 2231 | 2232 | 2233 | 2234 | 2235 | 2236 | 2237 | 2238 | 2239 |
| 8C | 2240 | 2241 | 2242 | 2243 | 2244 | 2245 | 2246 | 2247 | 2248 | 2249 | 2250 | 2251 | 2252 | 2253 | 2254 | 2255 |
| 8D | 2256 | 2257 | 2258 | 2259 | 2260 | 2261 | 2262 | 2263 | 2264 | 2265 | 2266 | 2267 | 2268 | 2269 | 2270 | 2271 |
| 8E | 2272 | 2273 | 2274 | 2275 | 2276 | 2277 | 2278 | 2279 | 2280 | 2281 | 2282 | 2283 | 2284 | 2285 | 2286 | 2287 |
| 8F | 2288 | 2289 | 2290 | 2291 | 2292 | 2293 | 2294 | 2295 | 2296 | 2297 | 2298 | 2299 | 2300 | 2301 | 2302 | 2303 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 90 | 2304 | 2305 | 2306 | 2307 | 2308 | 2309 | 2310 | 2311 | 2312 | 2313 | 2314 | 2315 | 2316 | 2317 | 2318 | 2319 |
| 91 | 2320 | 2321 | 2322 | 2323 | 2324 | 2325 | 2326 | 2327 | 2328 | 2329 | 2330 | 2331 | 2332 | 2333 | 2334 | 2335 |
| 92 | 2336 | 2337 | 2338 | 2339 | 2340 | 2341 | 2342 | 2343 | 2344 | 2345 | 2346 | 2347 | 2348 | 2349 | 2350 | 2351 |
| 93 | 2352 | 2353 | 2354 | 2355 | 2356 | 2357 | 2358 | 2359 | 2360 | 2361 | 2362 | 2363 | 2364 | 2365 | 2366 | 2367 |
| 94 | 2368 | 2369 | 2370 | 2371 | 2372 | 2373 | 2374 | 2375 | 2376 | 2377 | 2378 | 2379 | 2380 | 2381 | 2382 | 2383 |
| 95 | 2384 | 2385 | 2386 | 2387 | 2388 | 2389 | 2390 | 2391 | 2392 | 2393 | 2394 | 2395 | 2396 | 2397 | 2398 | 2399 |
| 96 | 2400 | 2401 | 2402 | 2403 | 2404 | 2405 | 2406 | 2407 | 2408 | 2409 | 2410 | 2411 | 2412 | 2413 | 2414 | 2415 |
| 97 | 2416 | 2417 | 2418 | 2419 | 2420 | 2421 | 2422 | 2423 | 2424 | 2425 | 2426 | 2427 | 2428 | 2429 | 2430 | 2431 |
| 98 | 2432 | 2433 | 2434 | 2435 | 2436 | 2437 | 2438 | 2439 | 2440 | 2441 | 2442 | 2443 | 2444 | 2445 | 2446 | 2447 |
| 99 | 2448 | 2449 | 2450 | 2451 | 2452 | 2453 | 2454 | 2455 | 2456 | 2457 | 2458 | 2459 | 2460 | 2461 | 2462 | 2463 |
| 9A | 2464 | 2465 | 2466 | 2467 | 2468 | 2469 | 2470 | 2471 | 2472 | 2473 | 2474 | 2475 | 2476 | 2477 | 2478 | 2479 |
| 9B | 2480 | 2481 | 2482 | 2483 | 2484 | 2485 | 2486 | 2487 | 2488 | 2489 | 2490 | 2491 | 2492 | 2493 | 2494 | 2495 |
| 9C | 2496 | 2497 | 2498 | 2499 | 2500 | 2501 | 2502 | 2503 | 2504 | 2505 | 2506 | 2507 | 2508 | 2509 | 2510 | 2511 |
| 9D | 2512 | 2513 | 2514 | 2515 | 2516 | 2517 | 2518 | 2519 | 2520 | 2521 | 2522 | 2523 | 2524 | 2525 | 2526 | 2527 |
| 9E | 2528 | 2529 | 2530 | 2531 | 2532 | 2533 | 2534 | 2535 | 2536 | 2537 | 2538 | 2539 | 2540 | 2541 | 2542 | 2543 |
| 9F | 2544 | 2545 | 2546 | 2547 | 2548 | 2549 | 2550 | 2551 | 2552 | 2553 | 2554 | 2555 | 2556 | 2557 | 2558 | 2559 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A0 | 2560 | 2561 | 2562 | 2563 | 2564 | 2565 | 2566 | 2567 | 2568 | 2569 | 2570 | 2571 | 2572 | 2573 | 2574 | 2575 |
| A1 | 2576 | 2577 | 2578 | 2579 | 2580 | 2581 | 2582 | 2583 | 2584 | 2585 | 2586 | 2587 | 2588 | 2589 | 2590 | 2591 |
| A2 | 2592 | 2593 | 2594 | 2595 | 2596 | 2597 | 2598 | 2599 | 2600 | 2601 | 2602 | 2603 | 2604 | 2605 | 2606 | 2607 |
| A3 | 2608 | 2609 | 2610 | 2611 | 2612 | 2613 | 2614 | 2615 | 2616 | 2617 | 2618 | 2619 | 2620 | 2621 | 2622 | 2623 |
| A4 | 2624 | 2625 | 2626 | 2627 | 2628 | 2629 | 2630 | 2631 | 2632 | 2633 | 2634 | 2635 | 2636 | 2637 | 2638 | 2639 |
| A5 | 2640 | 2641 | 2642 | 2643 | 2644 | 2645 | 2646 | 2647 | 2648 | 2649 | 2650 | 2651 | 2652 | 2653 | 2654 | 2655 |
| A6 | 2656 | 2657 | 2658 | 2659 | 2660 | 2661 | 2662 | 2663 | 2664 | 2665 | 2666 | 2667 | 2668 | 2669 | 2670 | 2671 |
| A7 | 2672 | 2673 | 2674 | 2675 | 2676 | 2677 | 2678 | 2679 | 2680 | 2681 | 2682 | 2683 | 2684 | 2685 | 2686 | 2687 |
| A8 | 2688 | 2689 | 2690 | 2691 | 2692 | 2693 | 2694 | 2695 | 2696 | 2697 | 2698 | 2699 | 2700 | 2701 | 2702 | 2703 |
| A9 | 2704 | 2705 | 2706 | 2707 | 2708 | 2709 | 2710 | 2711 | 2712 | 2713 | 2714 | 2715 | 2716 | 2717 | 2718 | 2719 |
| AA | 2720 | 2721 | 2722 | 2723 | 2724 | 2725 | 2726 | 2727 | 2728 | 2729 | 2730 | 2731 | 2732 | 2733 | 2734 | 2735 |
| AB | 2736 | 2737 | 2738 | 2739 | 2740 | 2741 | 2742 | 2743 | 2744 | 2745 | 2746 | 2747 | 2748 | 2749 | 2750 | 2751 |
| AC | 2752 | 2753 | 2754 | 2755 | 2756 | 2757 | 2758 | 2759 | 2760 | 2761 | 2762 | 2763 | 2764 | 2765 | 2766 | 2767 |
| AD | 2768 | 2769 | 2770 | 2771 | 2772 | 2773 | 2774 | 2775 | 2776 | 2777 | 2778 | 2779 | 2780 | 2781 | 2782 | 2783 |
| AE | 2784 | 2785 | 2786 | 2787 | 2788 | 2789 | 2790 | 2791 | 2792 | 2793 | 2794 | 2795 | 2796 | 2797 | 2798 | 2799 |
| AF | 2800 | 2801 | 2802 | 2803 | 2804 | 2805 | 2806 | 2807 | 2808 | 2809 | 2810 | 2811 | 2812 | 2813 | 2814 | 2815 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B0 | 2816 | 2817 | 2818 | 2819 | 2820 | 2821 | 2822 | 2823 | 2824 | 2825 | 2826 | 2827 | 2828 | 2829 | 2830 | 2831 |
| B1 | 2832 | 2833 | 2834 | 2835 | 2836 | 2837 | 2838 | 2839 | 2840 | 2841 | 2842 | 2843 | 2844 | 2845 | 2846 | 2847 |
| B2 | 2848 | 2849 | 2850 | 2851 | 2852 | 2853 | 2854 | 2855 | 2856 | 2857 | 2858 | 2859 | 2860 | 2861 | 2862 | 2863 |
| B3 | 2864 | 2865 | 2866 | 2867 | 2868 | 2869 | 2870 | 2871 | 2872 | 2873 | 2874 | 2875 | 2876 | 2877 | 2878 | 2879 |
| B4 | 2880 | 2881 | 2882 | 2883 | 2884 | 2885 | 2886 | 2887 | 2888 | 2889 | 2890 | 2891 | 2892 | 2893 | 2894 | 2895 |
| B5 | 2896 | 2897 | 2898 | 2899 | 2900 | 2901 | 2902 | 2903 | 2904 | 2905 | 2906 | 2907 | 2908 | 2909 | 2910 | 2911 |
| B6 | 2912 | 2913 | 2914 | 2915 | 2916 | 2917 | 2918 | 2919 | 2920 | 2921 | 2922 | 2923 | 2924 | 2925 | 2926 | 2927 |
| B7 | 2928 | 2929 | 2930 | 2931 | 2932 | 2933 | 2934 | 2935 | 2936 | 2937 | 2938 | 2939 | 2940 | 2941 | 2942 | 2943 |
| B8 | 2944 | 2945 | 2946 | 2947 | 2948 | 2949 | 2950 | 2951 | 2952 | 2953 | 2954 | 2955 | 2956 | 2957 | 2958 | 2959 |
| B9 | 2960 | 2961 | 2962 | 2963 | 2964 | 2965 | 2966 | 2967 | 2968 | 2969 | 2970 | 2971 | 2972 | 2973 | 2974 | 2975 |
| BA | 2976 | 2977 | 2978 | 2979 | 2980 | 2981 | 2982 | 2983 | 2984 | 2985 | 2986 | 2987 | 2988 | 2989 | 2990 | 2991 |
| BB | 2992 | 2993 | 2994 | 2995 | 2996 | 2997 | 2998 | 2999 | 3000 | 3001 | 3002 | 3003 | 3004 | 3005 | 3006 | 3007 |
| BC | 3008 | 3009 | 3010 | 3011 | 3012 | 3013 | 3014 | 3015 | 3016 | 3017 | 3018 | 3019 | 3020 | 3021 | 3022 | 3023 |
| BD | 3024 | 3025 | 3026 | 3027 | 3028 | 3029 | 3030 | 3031 | 3032 | 3033 | 3034 | 3035 | 3036 | 3037 | 3038 | 3039 |
| BE | 3040 | 3041 | 3042 | 3043 | 3044 | 3045 | 3046 | 3047 | 3048 | 3049 | 3050 | 3051 | 3052 | 3053 | 3054 | 3055 |
| BF | 3056 | 3057 | 3058 | 3059 | 3060 | 3061 | 3062 | 3063 | 3064 | 3065 | 3066 | 3067 | 3068 | 3069 | 3070 | 3071 |

A

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C0 | 3072 | 3073 | 3074 | 3075 | 3076 | 3077 | 3078 | 3079 | 3080 | 3081 | 3082 | 3083 | 3084 | 3085 | 3086 | 3087 |
| C1 | 3088 | 3089 | 3090 | 3091 | 3092 | 3093 | 3094 | 3095 | 3096 | 3097 | 3098 | 3099 | 3100 | 3101 | 3102 | 3103 |
| C2 | 3104 | 3105 | 3106 | 3107 | 3108 | 3109 | 3110 | 3111 | 3112 | 3113 | 3114 | 3115 | 3116 | 3117 | 3118 | 3119 |
| C3 | 3120 | 3121 | 3122 | 3123 | 3124 | 3125 | 3126 | 3127 | 3128 | 3129 | 3130 | 3131 | 3132 | 3133 | 3134 | 3135 |
| C4 | 3136 | 3137 | 3138 | 3139 | 3140 | 3141 | 3142 | 3143 | 3144 | 3145 | 3146 | 3147 | 3148 | 3149 | 3150 | 3151 |
| C5 | 3152 | 3153 | 3154 | 3155 | 3156 | 3157 | 3158 | 3159 | 3160 | 3161 | 3162 | 3163 | 3164 | 3165 | 3166 | 3167 |
| C6 | 3168 | 3169 | 3170 | 3171 | 3172 | 3173 | 3174 | 3175 | 3176 | 3177 | 3178 | 3179 | 3180 | 3181 | 3182 | 3183 |
| C7 | 3184 | 3185 | 3186 | 3187 | 3188 | 3189 | 3190 | 3191 | 3192 | 3193 | 3194 | 3195 | 3196 | 3197 | 3198 | 3199 |
| C8 | 3200 | 3201 | 3202 | 3203 | 3204 | 3205 | 3206 | 3207 | 3208 | 3209 | 3210 | 3211 | 3212 | 3213 | 3214 | 3215 |
| C9 | 3216 | 3217 | 3218 | 3219 | 3220 | 3221 | 3222 | 3223 | 3224 | 3225 | 3226 | 3227 | 3228 | 3229 | 3230 | 3231 |
| CA | 3232 | 3233 | 3234 | 3235 | 3236 | 3237 | 3238 | 3239 | 3240 | 3241 | 3242 | 3243 | 3244 | 3245 | 3246 | 3247 |
| CB | 3248 | 3249 | 3250 | 3251 | 3252 | 3253 | 3254 | 3255 | 3256 | 3257 | 3258 | 3259 | 3260 | 3261 | 3262 | 3263 |
| CC | 3264 | 3265 | 3266 | 3267 | 3268 | 3269 | 3270 | 3271 | 3272 | 3273 | 3274 | 3275 | 3276 | 3277 | 3278 | 3279 |
| CD | 3280 | 3281 | 3282 | 3283 | 3284 | 3285 | 3286 | 3287 | 3288 | 3289 | 3290 | 3291 | 3292 | 3293 | 3294 | 3295 |
| CE | 3296 | 3297 | 3298 | 3299 | 3300 | 3301 | 3302 | 3303 | 3304 | 3305 | 3306 | 3307 | 3308 | 3309 | 3310 | 3311 |
| CF | 3312 | 3313 | 3314 | 3315 | 3316 | 3317 | 3318 | 3319 | 3320 | 3321 | 3322 | 3323 | 3324 | 3325 | 3326 | 3327 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D0 | 3328 | 3329 | 3330 | 3331 | 3332 | 3333 | 3334 | 3335 | 3336 | 3337 | 3338 | 3339 | 3340 | 3341 | 3342 | 3343 |
| D1 | 3344 | 3345 | 3346 | 3347 | 3348 | 3349 | 3350 | 3351 | 3352 | 3353 | 3354 | 3355 | 3356 | 3357 | 3358 | 3359 |
| D2 | 3360 | 3361 | 3362 | 3363 | 3364 | 3365 | 3366 | 3367 | 3368 | 3369 | 3370 | 3371 | 3372 | 3373 | 3374 | 3375 |
| D3 | 3376 | 3377 | 3378 | 3379 | 3380 | 3381 | 3382 | 3383 | 3384 | 3385 | 3386 | 3387 | 3388 | 3389 | 3390 | 3391 |
| D4 | 3392 | 3393 | 3394 | 3395 | 3396 | 3397 | 3398 | 3399 | 3400 | 3401 | 3402 | 3403 | 3404 | 3405 | 3406 | 3407 |
| D5 | 3408 | 3409 | 3410 | 3411 | 3412 | 3413 | 3414 | 3415 | 3416 | 3417 | 3418 | 3419 | 3420 | 3421 | 3422 | 3423 |
| D6 | 3424 | 3425 | 3426 | 3427 | 3428 | 3429 | 3430 | 3431 | 3432 | 3433 | 3434 | 3435 | 3436 | 3437 | 3438 | 3439 |
| D7 | 3440 | 3441 | 3442 | 3443 | 3444 | 3445 | 3446 | 3447 | 3448 | 3449 | 3450 | 3451 | 3452 | 3453 | 3454 | 3455 |
| D8 | 3456 | 3457 | 3458 | 3459 | 3460 | 3461 | 3462 | 3463 | 3464 | 3465 | 3466 | 3467 | 3468 | 3469 | 3470 | 3471 |
| D9 | 3472 | 3473 | 3474 | 3475 | 3476 | 3477 | 3478 | 3479 | 3480 | 3481 | 3482 | 3483 | 3484 | 3485 | 3486 | 3487 |
| DA | 3488 | 3489 | 3490 | 3491 | 3492 | 3493 | 3494 | 3495 | 3496 | 3497 | 3498 | 3499 | 3500 | 3501 | 3502 | 3503 |
| DB | 3504 | 3505 | 3506 | 3507 | 3508 | 3509 | 3510 | 3511 | 3512 | 3513 | 3514 | 3515 | 3516 | 3517 | 3518 | 3519 |
| DC | 3520 | 3521 | 3522 | 3523 | 3524 | 3525 | 3526 | 3527 | 3528 | 3529 | 3530 | 3531 | 3532 | 3533 | 3534 | 3535 |
| DD | 3536 | 3537 | 3538 | 3539 | 3540 | 3541 | 3542 | 3543 | 3544 | 3545 | 3546 | 3547 | 3548 | 3549 | 3550 | 3551 |
| DE | 3552 | 3553 | 3554 | 3555 | 3556 | 3557 | 3558 | 3559 | 3560 | 3561 | 3562 | 3563 | 3564 | 3565 | 3566 | 3567 |
| DF | 3568 | 3569 | 3570 | 3571 | 3572 | 3573 | 3574 | 3575 | 3576 | 3577 | 3578 | 3579 | 3580 | 3581 | 3582 | 3583 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E0 | 3584 | 3585 | 3586 | 3587 | 3588 | 3589 | 3590 | 3591 | 3592 | 3593 | 3594 | 3595 | 3596 | 3597 | 3598 | 3599 |
| E1 | 3600 | 3601 | 3602 | 3603 | 3604 | 3605 | 3606 | 3607 | 3608 | 3609 | 3610 | 3611 | 3612 | 3613 | 3614 | 3615 |
| E2 | 3616 | 3617 | 3618 | 3619 | 3620 | 3621 | 3622 | 3623 | 3624 | 3625 | 3626 | 3627 | 3628 | 3629 | 3630 | 3631 |
| E3 | 3632 | 3633 | 3634 | 3635 | 3636 | 3637 | 3638 | 3639 | 3640 | 3641 | 3642 | 3643 | 3644 | 3645 | 3646 | 3647 |
| E4 | 3648 | 3649 | 3650 | 3651 | 3652 | 3653 | 3654 | 3655 | 3656 | 3657 | 3658 | 3659 | 3660 | 3661 | 3662 | 3663 |
| E5 | 3664 | 3665 | 3666 | 3667 | 3668 | 3669 | 3670 | 3671 | 3672 | 3673 | 3674 | 3675 | 3676 | 3677 | 3678 | 3679 |
| E6 | 3680 | 3681 | 3682 | 3683 | 3684 | 3685 | 3686 | 3687 | 3688 | 3689 | 3690 | 3691 | 3692 | 3693 | 3694 | 3695 |
| E7 | 3696 | 3697 | 3698 | 3699 | 3700 | 3701 | 3702 | 3703 | 3704 | 3705 | 3706 | 3707 | 3708 | 3709 | 3710 | 3711 |
| E8 | 3712 | 3713 | 3714 | 3715 | 3716 | 3717 | 3718 | 3719 | 3720 | 3721 | 3722 | 3723 | 3724 | 3725 | 3726 | 3727 |
| E9 | 3728 | 3729 | 3730 | 3731 | 3732 | 3733 | 3734 | 3735 | 3736 | 3737 | 3738 | 3739 | 3740 | 3741 | 3742 | 3743 |
| EA | 3744 | 3745 | 3746 | 3747 | 3748 | 3749 | 3750 | 3751 | 3752 | 3753 | 3754 | 3755 | 3756 | 3757 | 3758 | 3759 |
| EB | 3760 | 3761 | 3762 | 3763 | 3764 | 3765 | 3766 | 3767 | 3768 | 3769 | 3770 | 3771 | 3772 | 3773 | 3774 | 3775 |
| EC | 3776 | 3777 | 3778 | 3779 | 3780 | 3781 | 3782 | 3783 | 3784 | 3785 | 3786 | 3787 | 3788 | 3789 | 3790 | 3791 |
| ED | 3792 | 3793 | 3794 | 3795 | 3796 | 3797 | 3798 | 3799 | 3800 | 3801 | 3802 | 3803 | 3804 | 3805 | 3806 | 3807 |
| EE | 3808 | 3809 | 3810 | 3811 | 3812 | 3813 | 3814 | 3815 | 3816 | 3817 | 3818 | 3819 | 3820 | 3821 | 3822 | 3823 |
| EF | 3824 | 3825 | 3826 | 3827 | 3828 | 3829 | 3830 | 3831 | 3832 | 3833 | 3834 | 3835 | 3836 | 3837 | 3838 | 3839 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| F0 | 3840 | 3841 | 3842 | 3843 | 3844 | 3845 | 3846 | 3847 | 3848 | 3849 | 3850 | 3851 | 3852 | 3853 | 3854 | 3855 |
| F1 | 3856 | 3857 | 3858 | 3859 | 3860 | 3861 | 3862 | 3863 | 3864 | 3865 | 3866 | 3867 | 3868 | 3869 | 3870 | 3871 |
| F2 | 3872 | 3873 | 3874 | 3875 | 3876 | 3877 | 3878 | 3879 | 3880 | 3881 | 3882 | 3883 | 3884 | 3885 | 3886 | 3887 |
| F3 | 3888 | 3889 | 3890 | 3891 | 3892 | 3893 | 3894 | 3895 | 3896 | 3897 | 3898 | 3899 | 3900 | 3901 | 3902 | 3903 |
| F4 | 3904 | 3905 | 3906 | 3907 | 3908 | 3909 | 3910 | 3911 | 3912 | 3913 | 3914 | 3915 | 3916 | 3917 | 3918 | 3919 |
| F5 | 3920 | 3921 | 3922 | 3923 | 3924 | 3925 | 3926 | 3927 | 3928 | 3929 | 3930 | 3931 | 3932 | 3933 | 3934 | 3935 |
| F6 | 3936 | 3937 | 3938 | 3939 | 3940 | 3941 | 3942 | 3943 | 3944 | 3945 | 3946 | 3947 | 3948 | 3949 | 3950 | 3951 |
| F7 | 3952 | 3953 | 3954 | 3955 | 3956 | 3957 | 3958 | 3959 | 3960 | 3961 | 3962 | 3963 | 3964 | 3965 | 3966 | 3967 |
| F8 | 3968 | 3969 | 3970 | 3971 | 3972 | 3973 | 3974 | 3975 | 3976 | 3977 | 3978 | 3979 | 3980 | 3981 | 3982 | 3983 |
| F9 | 3984 | 3985 | 3986 | 3987 | 3988 | 3989 | 3990 | 3991 | 3992 | 3993 | 3994 | 3995 | 3996 | 3997 | 3998 | 3999 |
| FA | 4000 | 4001 | 4002 | 4003 | 4004 | 4005 | 4006 | 4007 | 4008 | 4009 | 4010 | 4011 | 4012 | 4013 | 4014 | 4015 |
| FB | 4016 | 4017 | 4018 | 4019 | 4020 | 4021 | 4022 | 4023 | 4024 | 4025 | 4026 | 4027 | 4028 | 4029 | 4030 | 4031 |
| FC | 4032 | 4033 | 4034 | 4035 | 4036 | 4037 | 4038 | 4039 | 4040 | 4041 | 4042 | 4043 | 4044 | 4045 | 4046 | 4047 |
| FD | 4048 | 4049 | 4050 | 4051 | 4052 | 4053 | 4054 | 4055 | 4056 | 4057 | 4058 | 4059 | 4060 | 4061 | 4062 | 4063 |
| FE | 4064 | 4065 | 4066 | 4067 | 4068 | 4069 | 4070 | 4071 | 4072 | 4073 | 4074 | 4075 | 4076 | 4077 | 4078 | 4079 |
| FF | 4080 | 4081 | 4082 | 4083 | 4084 | 4085 | 4086 | 4087 | 4088 | 4089 | 4090 | 4091 | 4092 | 4093 | 4094 | 4095 |

A

## A.5 ASSEMBLER ERROR FLAGS

The assembler will put only the letter on a line which contains an error. At the end of the assembly, all of the error flags are broken down by letter and number, a brief explanation is given, and cross-reference information is given for each detailed flag.

| | CODE | T E X T | Severity code |
|---|---|---|---|
| A | A10 | RELOCATABLE TERM IN PRODUCT OR DIVISION | 1 |
| | A11 | COMPLEX-RELOCATABLE EXPRESSION IN EQU-STATEMENT | 1 |
| | A12 | EQU-STATEMENT IN XDSECT | 1 |
| | A13 | RELOCATABILITY ERROR - NEGATIVE EXPRESSION | 1 |
| | A14 | RELOCATABILITY ERROR IN COMPLEX RELOCATABLE EXPRESSION | 1 |
| | A15 | RELOCATABILITY ERROR | 1 |
| | A20 | ADDRESS GREATER THAN 2**24-1; NEAR OPERAND COLUMN NN | 1 |
| | A21 | NEGATIVE ADDRESS; NEAR OPERAND COLUMN NN | 1 |
| B | B10 | INVALID OPERAND IN ICTL OR ISEQ STATEMENT | 1 |
| | B11 | ICTL APPEARS TOO LATE | 1 |
| | B20 | ILLEGAL START STATEMENT | 0 |
| | B30 | INVALID OPERAND IN START STATEMENT | 1 |
| | B31 | ILLEGAL ATTRIBUTE IN CSECT OR START STATEMENT | 1 |
| | B40 | INVALID OPCODE IN OPSYN OPERAND OR NAME FIELD | 1 |
| | B41 | OPSYN IN MACRO NOT ALLOWED | 1 |
| | B42 | COPY ELEMENT NOT FOUND | 1 |
| | | | |

A

| (B) | CODE | T E X T | Severity code |
|---|---|---|---|
| | B43 | COPY STATEMENT OPERAND NOT A VALID SYMBOL | 1 |
| | B44 | IN MACROLIBRARY PROTO-CALL IS DIFFERENT TO ENTRY NAME | 0 |
| | B50 | UNEXPECTED EOF ON SYSTEM INPUT | 0 |
| | B51 | END STATEMENT READ BEFORE A MEND STATEMENT WAS ENCOUNTERED | 0 |
| | B52 | RECORD TRUNCATED | 0 |
| C | C10 | ILLEGAL SYMBOLIC PARAMETER; XXXXXXXX | 1 |
| | C20 | SYMBOLIC PARAMETER PREVIOUSLY DEFINED | 1 |
| | C30 | SOURCE - MACRO HAS BEEN PREVIOUSLY DEFINED; XXXXXXXX | 1 |
| | C40 | MACRO INSTRUCTION PROTOTYPE STATEMENT HAS INVALID OPCODE | 1 |
| | C51 | INVALID FORMAT IN PROTOTYPE PARAMETER; NEAR OPERAND COLUMN NN | 1 |
| | C52 | INVALID FORMAT IN MACRO INSTRUCTION | 1 |
| | C53 | INVALID FORMAT IN GSEQ STATEMENT | 1 |
| | C54 | INCONSISTENT MACRO CALL; NEAR OPERAND COLUMN NN | 0 |
| | C55 | INVALID MACRO NAME IN MCALL | 1 |
| | C60 | PARAMETER IN MACRO PROTOTYPE OR MACRO INSTRUCTION IS TOO LONG | 1 |
| | C61 | OPERAND IN MACRO CALL IS TOO LONG | 1 |
| | C70 | POSITIONAL MACRO OPERAND APPEARS AFTER KEYWORD OPERAND | 1 |
| | C80 | UNDEFINED OR MULTIPLY SPECIFIED KEYWORD IN MACRO CALL; XXXXXXXX | 1 |
| D | D1 | INVALID CONSTANT TYPE | 1 |
| | | | |

| | CODE | T E X T | Severity code |
|---|---|---|---|
| (D) | D2 | LENGTH MODIFIER ERROR; NEAR OPERAND COLUMN NN | 1 |
| | D3 | S-TYPE CONSTANT IN LITERAL STRING | 1 |
| | D4 | QUOTES NOT PAIRED OR ILLEGAL TERMINATION OF QUOTED STRING | 1 |
| | D5 | MISSING OPERAND | 1 |
| | D6 | STATEMENT OUT OF SEQUENCE | 0 |
| | D7 | REFERENCED ADDRESS ALIGNMENT ERROR; NEAR OPERAND COLUMN NN | 1 |
| | D8 | SINGLE AMPERSAND OR QUOTE IN STRING; NEAR OPERAND COLUMN NN | 1 |
| | D9 | INVALID DISPLACEMENT | 1 |
| | D10 | INVALID REGISTER SPECIFICATION | 1 |
| | D11 | INVALID LENGTH SPECIFICATION; NEAR OPERAND COLUMN NN | 1 |
| | D12 | DUPLICATION FACTOR ERROR | 1 |
| | D13 | SCALE MODIFIER ERROR | 1 |
| | D14 | EXPONENT MODIFIER ERROR | 1 |
| | D15 | PRECISION LOST - CONSTANT TRUNCATED; NEAR OPERAND COLUMN NN | 1 |
| | D16 | DATA ITEM OF SELFDEFINING TERM TOO LARGE | 1 |
| | D17 | ARITHMETIC OVERFLOW; NEAR OPERAND COLUMN NN | 1 |
| | D18 | FLOATING POINT CHARACTERISTIC OUT OF RANGE | 1 |
| | D19 | INVALID CHARACTER IN CONSTANT; NEAR OPERAND COLUMN NN | 1 |
| | D20 | INVALID OPERAND IN Q-CONSTANT | 1 |

A

| | CODE | T E X T | Severity code |
|---|---|---|---|
| (D) | D21 | EXPRESSION OF EQU OPERAND IS NEGATIVE OR EXCEEDS 2**24-1 | 1 |
| | D30 | INVALID REGISTER SPECIFICATION - EVEN NUMBERED REGISTER REQUIRED | 2 |
| | D31 | INVALID REGISTER SPECIFICATION - FLOATING POINT REGISTER REQUIRED | 2 |
| | D32 | INVALID REGISTER SPECIFICATION | 2 |
| | D33 | REGISTER OVERFLOW OR RELOCATABLE | 2 |
| | D51 | REQUIRED OPERAND FIELD MISSING | 1 |
| | D91 | DISPLACEMENT GREATER THAN X'FFF' | 1 |
| E | E1 | INVALID DELIMITER; NEAR OPERAND COLUMN NN | 1 |
| | E9 | INVALID CCW | 2 |
| | E10 | ILLEGAL CONTINUATION | 1 |
| | E11 | EOF REACHED WHEN CONTINUING LINE WAS EXPECTED | 1 |
| | E12 | QUOTE MISSING; NEAR OPERAND COLUMN NN | 1 |
| | E13 | NULL STRING; NEAR OPERAND COLUMN NN | 1 |
| | E14 | OPERAND SHOULD BEGIN WITH A QUOTE | 1 |
| | E15 | ILLEGAL DELIMITER - RIGHT PARENTHESIS EXPECTED; NEAR OPERAND COLUMN NN | 1 |
| | E16 | TOO MANY STACK OR UNSTACK STATEMENTS | 1 |
| | E20 | INVALID SYMBOL; XXXXXXXX | 1 |
| | E21 | MISSING OR INVALID SYMBOL IN NAME FIELD | 1 |
| | E22 | AGO OR AIF OPERAND NOT A SEQUENCE SYMBOL | 1 |
| | | | |

| | CODE | T E X T | Severity code |
|---|---|---|---|
| (E) | E23 | INVALID SEQUENCE SYMBOL IN GSEQ STATEMENT | 1 |
| | E30 | INVALID SYNTAX IN EXPRESSION; NEAR OPERAND COLUMN NN | 1 |
| | E31 | ARITHMETIC TERM COULD NOT BE SOLVED; NEAR OPERAND COLUMN NN | 1 |
| | E32 | INVALID SYNTAX IN SPACE OR EJECT STATEMENT | 0 |
| | E33 | INVALID USING OR DROP STATEMENT | 2 |
| | E34 | DROP ISSUED FOR A REGISTER THAT WAS NOT ASSIGNED BY USING STATEMENT | 0 |
| | E40 | TOO MANY LEVELS OF PARENTHESES | 1 |
| | E41 | MORE THAN 6 LEVELS OF PARENTHESES; NEAR OPERAND COLUMN NN | 1 |
| | E50 | MISSING RIGHT PARENTHESIS; NEAR OPERAND COLUMN NN | 1 |
| | E51 | UNPAIRED PARENTHESES OR BLANK FOUND INSIDE PAIRED PARENTHESES | 1 |
| | E52 | TOO MANY RIGHT PARENTHESES | 1 |
| | E60 | INVALID OPERAND IN PRINT OR TITLE STATEMENT; NEAR OPERAND COLUMN NN | 0 |
| | E61 | INVALID OPERAND IN STACK OR UNSTACK STATEMENT | 1 |
| | E70 | INVALID LITERAL USAGE; NEAR OPERAND COLUMN NN | 1 |
| | E71 | ILLEGAL LITERAL USAGE; NEAR OPERAND COLUMN NN | 2 |
| | E80 | TITLE STATEMENT OPERAND EXCEEDS 100 CHARACTERS | 0 |
| G | G10 | ILLEGAL STATEMENT OUTSIDE OF MACRO DEFINITION | 1 |
| | G11 | ILLEGAL MACRO, MEND OR MEXIT STATEMENT - ONLY ALLOWED WITHIN MACRO DEFINITION | 1 |
| | G12 | .* COMMENT STATEMENT IS ILLEGAL OUTSIDE OF MACRO DEFINITION | 0 |
| | | | |

A

| | CODE | T E X T | Severity code |
|---|---|---|---|
| (G) | G20 | ILLEGAL STATEMENT INSIDE OF MACRO DEFINITION | 1 |
| | G21 | OPSYN OR MACRO MAY NOT APPEAR INSIDE MACRO DEFINITION | 1 |
| | G22 | ISEQ IS ILLEGAL IN COPY CODE | 1 |
| | G30 | OPERATION NOT ALLOWED TO BE GENERATED | 1 |
| | G31 | GENERATED OPCODE INVALID OR UNDEFINED | 1 |
| | G32 | GENERATED OPCODE IS BLANK | 1 |
| | G33 | GENERATED OPCODE EXCEEDS 8 CHARACTERS | 1 |
| H | H10 | LOCATION COUNTER OVERFLOW | 1 |
| | H11 | DXD LOCATION COUNTER OVERFLOW | 1 |
| I | I10 | INVALID IMMEDIATE FIELD | 1 |
| | I20 | INVALID SELFDEFINING TERM; NEAR OPERAND COLUMN NN | 1 |
| K | K10 | SEQUENCE SYMBOL MULTIPLY DEFINED; XXXXXX | 0 |
| L | L10 | INVALID LTORG STATEMENT | 1 |
| | L20 | ESD TABLE OVERFLOW | 2 |
| | L30 | NAME OF STATEMENT IN DSECT USED IN RELOCATABLE ADDRESS CONSTANT | 1 |
| | L40 | TOO MANY SECTIONS | 2 |
| | L41 | TOO MANY LTORGS | 1 |
| | L42 | LTORG INVALID AT THIS TIME | 0 |
| | L50 | OPERAND 2 OF CNOP MUST BE 4 OR 8 | 2 |

| | CODE | T E X T | Severity code |
|---|---|---|---|
| (L) | L51 | OPERAND 1 OF CNOP MUST BE 0, 2, 4, 6 OR 8 | 2 |
| | L90 | MORE THAN 5 ERRORS IN THIS STATEMENT | 0 |
| M | M10 | MULTIPLY DEFINED SYMBOL; XXXXXXXX | 1 |
| | M20 | ENTRY POINT IN DSECT, COM, OR UNNAMED CSECT | 1 |
| | M30 | SYMBOL DEFINITION EXCEEDS 2**24-1 | 1 |
| | M31 | SECOND OPERAND VALUE IS OUTSIDE ALLOWABLE RANGE | 1 |
| | M40 | ENTRY NAME REFERS EXTRN VALUE | 1 |
| | M50 | CSECT, DSECT, OR COM WITH SAME NAME | 1 |
| | M51 | DXD-NAME PREVIOUSLY DEFINED | 1 |
| | M55 | UNNAMED DSECT | 1 |
| N | N10 | UNDEFINED VARIABLE SYMBOL; XXXXXXXX | 1 |
| | N20 | UNDEFINED SEQUENCE SYMBOL; XXXXXXXX | 1 |
| | N30 | ILLEGAL OCCURENCE OF GBL, LCL OR GSEQ | 1 |
| | N31 | ISEQ APPEARS IN MACRO DEFINITION | 0 |
| O | O2 | INVALID OPCODE | 1 |
| | O21 | OPCODE WILL NOT BE SUPPORTED BY BS2000 WITHIN A FORESEEABLE SPACE OF TIME | 0 |
| | O23 | OPCODE WILL BE SUPPORTED BY BS2000 WITHIN A FORESEEABLE SPACE OF TIME | 0 |
| P | P1 | PRIVILEGED INSTRUCTION | 0 |
| Q | Q10 | INVALID ORIGIN STATEMENT | 1 |

A

| | CODE | T E X T | Severity code |
|---|---|---|---|
| (Q) | Q11 | SYMBOL IN ORG OPERAND DOES NOT BELONG TO CURRENT SECTION | 1 |
| | Q12 | ORG OPERAND IS ABSOLUTE - MUST BE RELOCATABLE | 1 |
| | Q20 | SYMBOL NOT PREVIOUSLY DEFINED OR IN XDSEC | 1 |
| R | R10 | SUBSCRIPT OUT OF RANGE | 1 |
| | R11 | INVALID SUBSCRIPT | 1 |
| S | S10 | ILLEGAL NAME FIELD | 1 |
| | S11 | GENERATED NAME FIELD ENTRY IS INVALID | 1 |
| | S12 | MORE THAN 1 TITLE STATEMENT NAMED | 0 |
| T | T10 | GENERATED STRING TOO LONG | 1 |
| | T11 | GENERATED FIELD EXCEEDS 255 CHARACTERS | 1 |
| | T12 | GENERATED SYMBOL IN NAME FIELD EXCEEDS 8 CHARACTERS | 1 |
| U | U10 | UNDEFINED SYMBOL; XXXXXXXX | 1 |
| | U11 | INVALID ENTRY OPERAND - LINKAGE CANNOT BE PERFORMED | 1 |
| V | V10 | SET SYMBOL PREVIOUSLY DEFINED | 1 |
| | V20 | VARIABLE SYMBOL MATCHES PARAMETER IN SAME MACRO DEFINITION | 1 |
| | V30 | INCONSISTENT GLOBAL DECLARATION; XXXXXXXX | 1 |
| | V40 | ILLEGAL USE OF SYSTEM VARIABLE SYMBOL; XXXXXXX | 1 |
| | V41 | SYMBOLIC PARAMETER IN GLOBAL OR LOCAL DECLARATION IS SYSTEM VARIABLE SYMBOL; XXXXXXXX | 1 |
| | V42 | SYMBOLIC PARAMETER USED IN MACRO PROTOTYPE IS SYSTEM VARIABLE SYMBOL; XXXXXXXX | 1 |

| | CODE | T E X T | Severity code |
|---|---|---|---|
| (V) | V43 | SYMBOLIC PARAMETER BEGINS WITH RESERVED STRING '&SYS' | 0 |
| | V50 | INVALID FORMAT FOR VARIABLE SYMBOL | 1 |
| | V51 | INVALID USE OF SYSTEM VARIABLE &SYSECT | 1 |
| | V52 | INVALID USE OF A DIMENSIONED VARIABLE | 1 |
| | V60 | NAME FIELD CONTAINS ILLEGAL SET SYMBOL | 1 |
| | V61 | SYMBOL IN NAME FIELD OF OPSYN STATEMENT SHOULD NOT BE CONCATENATED | 1 |
| W | W10 | OPERAND 2 OF SS-INSTRUCTION USES REGISTER 0 FOR BASEREGISTER | 0 |
| | W11 | OPERAND 2 OF INSTRUCTION IS SELFDEFINING | 0 |
| X | X10 | SINGLE TERM LOGICAL EXPRESSION IS NOT A SETB SYMBOL | 1 |
| | X20 | ILLEGAL ATTRIBUTE NOTATION; NEAR OPERAND COLUMN NN | 1 |
| | X21 | ATTRIBUTE REFERS ILLEGAL VARIABLE SYMBOL | 1 |
| | X30 | INVALID SUBSCRIPT; NEAR OPERAND COLUMN NN | 1 |
| | X31 | SUBSCRIPT EXPRESSION EXCEEDS MAXIMUM DIMENSION | 1 |
| | X32 | INVALID DUPLICATION FACTOR | 1 |
| | X33 | SUBSCRIPT IS 0, NEGATIV OR GREATER THAN 127 | 1 |
| | X40 | ILLEGAL OPERAND FIELD FORMAT; NEAR OPERAND COLUMN NN | 1 |
| | X41 | INVALID SEVERITY CODE IN MNOTE STATEMENT | 1 |
| | X42 | SYMBOLIC VARIABLE IN CHARACTER STRING | 1 |
| | X43 | ARITHMETIC OVERFLOW IN INTERMEDIATE RESULT | 1 |

A

| | CODE | T E X T | Severity code |
|---|---|---|---|
| (X) | X44 | OPERATOR ERROR IN AIF STATEMENT | 1 |
| | X45 | CHARACTER STRING USED AS ARITHMETIC TERM CONTAINS NONDECIMAL CHARACTERS | 1 |
| | X46 | OPERAND USED IN BOOLEAN RELATION IS NOT BOOLEAN | 1 |
| | X47 | OPERAND ERROR IN RELATION | 1 |
| Y | Y10 | REFERENCED ADDRESS NOT IN RANGE OF USING STATEMENT | 1 |
| Z | Z10 | SYMBOL TABLE OVERFLOW | 3 |
| | Z20 | GLOBAL DIRECTORY OVERFLOW | 3 |
| | Z30 | LOCAL DIRECTORY OVERFLOW | 3 |
| | Z40 | ACTR EXCEEDED IN MACRO | 1 |
| | Z41 | ACTR EXCEEDED IN SOURCE | 3 |
| | Z50 | ATTRIBUTE LIST OVERFLOW | 3 |
| | Z60 | LITERAL POOL OVERFLOW | 3 |

## A.6 INSTRUCTION FORMAT (SET1)

Instructions for processors 4004/150, 4004/220, 7.720, 7.730, 7.738, 7.740, 7.748, 7.750, 7.755, 7.760, 7.762 and all 7.500

| Mnemonic Code | Instruction Name | Mach. Code | Length | Operand Format |
|---|---|---|---|---|
| A | Add (word) | 5A | 4 | R1,D2(X2,B2) |
| AD | Add normalized, long | 6A | 4 | R1,D2(X2,B2) |
| ADR | Add normalized, long (register) | 2A | 2 | R1,R2 |
| AE | Add normalized, short | 7A | 4 | R1,D2(X2,B2) |
| AER | Add normalized, short (register) | 3A | 2 | R1,R2 |
| AH | Add halfword | 4A | 4 | R1,D2(X2,B2) |
| AL | Add logical | 5E | 4 | R1,D2(X2,B2) |
| ALR | Add logical (register) | 1E | 2 | R1,R2 |
| AP | Add decimal | FA | 6 | D1(L1,B1),D2(L2,B2) |
| AR | Add (register) | 1A | 2 | R1,R2 |
| AU | Add unnormalized, short | 7E | 4 | R1,D2(X2,B2) |
| AUR | Add unnormalized, short (register) | 3E | 2 | R1,R2 |
| AW | Add unnormalized, long | 6E | 4 | R1,D2(X2,B2) |
| AWR | Add unnormalized, long (register) | 2E | 2 | R1,R2 |
| AXR | Add normalized, extended | 36 | 2 | R1,R2 |
| BAL | Branch and load register | 45 | 4 | R1,D2(X2,B2) |
| BALR | Branch and load register | 05 | 2 | R1,R2 |
| 2)BAS | Branch and store register | 4D | 4 | R1,D2(X2,B2) |
| 2)BASR | Branch and store register | 0D | 2 | R1,R2 |
| BC | Branch on condition | 47 | 4 | I,D2(X2,B2) |
| BCR | Branch on condition (register) | 07 | 2 | I,R2 |
| BCT | Branch on count | 46 | 4 | R1,D2(X2,B2) |
| BCTR | Branch on count (register) | 06 | 2 | R1,R2 |
| BXH | Branch on index high | 86 | 4 | R1,R3,D2(B2) |
| BXLE | Branch on index low or equal | 87 | 4 | R1,R3,D2(B2) |
| C | Compare (word) | 59 | 4 | R1,D2(X2,B2) |
| * CCPU | Check CPU | AC | 4 | D1(B1),I2 |
| CD | Compare, long | 69 | 4 | R1,D2(X2,B2) |
| CDR | Compare, long (register) | 29 | 2 | R1,R2 |
| CDS | Compare double and swap | BB | 4 | R1,R3,D2(B2) |
| CE | Compare, short | 79 | 4 | R1,D2(X2,B2) |
| CER | Compare, short (register) | 39 | 2 | R1,R2 |
| CH | Compare halfword | 49 | 4 | R1,D2(C2,B2) |
| * CIOC | Check I/O Control | AD | 4 | D1(B1),I2 |
| * CKC | Check channel | 9F | 4 | D1(B1) |
| CL | Compare logical (word) | 55 | 4 | R1,D2(X2,B2) |
| CLC | Compare logical | D5 | 6 | D1(L,B1),D2(B2) |
| CLCL | Compare logical, long | 0F | 2 | R1,R2 |
| CLI | Compare logical immediate | 95 | 4 | D1(B1),I2 |
| CLM | Compare logical under mask | BD | 4 | R1,M3,D2(B2) |
| CLR | Compare logical (register) | 15 | 2 | R1,R2 |
| CP | Compare decimal | F9 | 6 | D1(L1,B1),D2(L2,B2) |
| CR | Compare (register) | 19 | 2 | R1,R2 |
| CS | Compare and swap | BA | 4 | R1,R3,D2(B2) |
| CVB | Convert to binary | 4F | 4 | R1,D2(X2,B2) |
| CVD | Convert to decimal | 4E | 4 | R1,D2(X2,B2) |
| D | Divide (word) | 5D | 4 | R1,D2(X2,B2) |
| DD | Divide, long | 6D | 4 | R1,D2(X2,B2) |
| DDR | Divide, long (register) | 2D | 2 | R1,R2 |
| DE | Divide, short | 7D | 4 | R1,D2(X2,B2) |
| DER | Divide, short (register) | 3D | 2 | R1,R2 |
| * DIG | Diagnose | 83 | 4 | D1(B1) |

A

| Mnemonic Code | Instruction Name | Mach. Code | Length | Operand Format |
|---|---|---|---|---|
| DP | Divide decimal | FD | 6 | D1(L1,B1),D2(L2,B2) |
| DR | Divide (register) | 1D | 2 | R1,R2 |
| ED | Edit | DE | 6 | D1(L,B1),D2(B2) |
| EDMK | Edit and mark | DF | 6 | D1(L,B1),D2(B2) |
| EX | Execute | 44 | 4 | R1,D2(X2,B2) |
| 1) EXST | Execute stack | B1 | 4 | R1,M3,D2(B2) |
| * FC | Function call | 9A | 4 | D1(B1),I2 (for 4004/151) |
| * FCAL | Function call | B7 | 4 | D1(B1),I2 (for 4004/220-230) |
| HDR | Halve, long (register) | 24 | 2 | R1,R2 |
| * HDV | Halt device | 9E | 4 | D1(B1) |
| HER | Halve, short (register) | 34 | 2 | R1,R2 |
| IC | Insert character | 43 | 4 | R1,D2(X2,B2) |
| ICM | Insert character under mask | BF | 4 | R1,M3,D2(B2) |
| * IDL | Idle | 80 | 4 | I2 |
| * ISK | Insert storage key | 09 | 2 | R1,R2 |
| L | Load (word) | 58 | 4 | R1,D2(X2,B2) |
| LA | Load address | 41 | 4 | R1,D2(X2,B2) |
| 1) LBF | Load bit field | B6 | 4 | B1(D1),D3(L3) |
| LCDR | Load complement, long (register) | 23 | 2 | R1,R2 |
| LCER | Load complement, short (register) | 33 | 2 | R1,R2 |
| LCR | Load complement | 13 | 2 | R1,R2 |
| LD | Load, long | 68 | 4 | R1,D2(X2,B2) |
| LDR | Load, long (register) | 28 | 2 | R1,R2 |
| LE | Load, short | 78 | 4 | R1,D2(X2,B2) |
| LER | Load, short (register) | 38 | 2 | R1,R2 |
| LH | Load halfword | 48 | 4 | R1,D2(X2,B2) |
| LM | Load multiple | 98 | 4 | R1,R3,D2(B2) |
| LNDR | Load negative, long (register) | 21 | 2 | R1,R2 |
| LNER | Load negative, short (register) | 31 | 2 | R1,R2 |
| LNR | Load negative | 11 | 2 | R1,R2 |
| LPDR | Load positive, long (register) | 20 | 2 | R1,R2 |
| LPER | Load positive, short | 30 | 2 | R1,R2 |
| LPR | Load positive | 10 | 2 | R1,R2 |
| LR | Load (word) | 18 | 2 | R1,R2 |
| LRDR | Load and round, extended, result long | 25 | 2 | R1,R2 |
| LRER | Load and round, extended, result short | 35 | 2 | R1,R2 |
| * LSM | Load shaded memory | D9 | 6 | D1(L,B1),D2(B2) |
| * LSP | Load scratch pad | D8 | 6 | D1(L,B1),D2(B2) |
| LTDR | Load and test, long (register) | 22 | 2 | R1,R2 |
| LTER | Load and test, short (register) | 32 | 2 | R1,R2 |
| LTR | Load and test | 12 | 2 | R1,R2 |
| 1) LWI | Load word indirect | 53 | 4 | R1,D2(X2,B2) |
| M | Multiply (word) | 5C | 4 | R1,D2(X2,B2) |
| MD | Multiply, long | 6C | 4 | R1,D2(X2,B2) |
| MDR | Multiply, long (register) | 2C | 2 | R1,R2 |
| ME | Multiply, short | 7C | 4 | R1,D2(X2,B2) |
| MER | Multiply, short (register) | 3C | 2 | R1,R2 |
| MH | Multiply halfword | 4C | 4 | R1,D2(X2,B2) |
| MP | Multiply halfword | FC | 6 | D1(L1,B1),D2(L2,B2) |
| MR | Multiply (register) | 1C | 2 | R1,R2 |
| MVC | Move | D2 | 6 | D1(L,B1),D2(B2) |

| Mnemonic Code | Instruction Name | Mach. Code | Length | Operand Format |
|---|---|---|---|---|
| MVCL | Move, long | 0E | 2 | R1,R2 |
| MVI | Move immediate | 92 | 4 | D1(B1),I2 |
| MVN | Move numerics | D1 | 6 | D1(L,B1),D2(B2) |
| MVO | Move with offset | F1 | 6 | D1(L1,B1),D2(L2,B2) |
| MVZ | Move zones | D3 | 6 | D1(L,B1),D2(B2) |
| MXD | Multiply, extended, result long | 67 | 4 | R1,D2(X2,B2) |
| MXDR | Multiply, extended, result long | 27 | 2 | R1,R2 |
| MXR | Multiply, extended, result short | 26 | 2 | R1,R2 |
| N | AND (word) | 54 | 4 | R1,D2(X2,B2) |
| NC | AND (characters) | D4 | 6 | D1(L,B1),D2(B2) |
| NI | AND immediate | 94 | 4 | D1(B1),I2 |
| NR | AND (register) | 14 | 2 | R1,R2 |
| O | OR (word) | 56 | 4 | R1,D2(X2,B2) |
| OC | OR (characters) | D6 | 6 | D1(L,B1),D2(B2) |
| OI | OR immediate | 96 | 4 | D1(B1),I2 |
| OR | OR (register) | 16 | 2 | R1,R2 |
| PACK | Pack | F2 | 6 | D1(L1,B1),D2(L2,B2) |
| * PC | Program control | 82 | 4 | D1(B1),I2 |
| 1)POP | Pop | 9B | 4 | R1,R3,D2(B2) |
| 1)PUSH | Push | 99 | 4 | R1,R3,D2(B2) |
| * RDD | Read direct | 85 | 4 | D1(B1),I2 |
| S | Subtract (word) | 5B | 4 | R1,D2(X2,B2) |
| SD | Subtract normalized, long | 6B | 4 | R1,D2(X2,B2) |
| SDR | Subtract normalized, long (register) | 2B | 2 | R1,R2 |
| * SDV | Start device | 9C | 4 | D1(B1) |
| SE | Subtract normalized, short | 7B | 4 | R1,D2(X2,B2) |
| SER | Subtract normalized, short (register) | 3B | 2 | R1,R2 |
| SH | Subtract halfword | 4B | 4 | R1,D2(X2,B2) |
| SL | Subtract logical | 5F | 4 | R1,D2(X2,B2) |
| SLA | Shift left single | 8B | 4 | R1,D2(B2) |
| SLDA | Shift left double | 8F | 4 | R1,D2(B2) |
| SLDL | Shift left double logical | 8D | 4 | R1,D2(B2) |
| SLL | Shift left single logical | 89 | 4 | R1,D2(B2) |
| SLR | Subtract logical | 1F | 2 | R1,R2 |
| SP | Subtract decimal | FB | 6 | D1(L1,B1),D2(L2,B2) |
| SPM | SET program mask | 04 | 2 | R1 |
| SR | Subtract (register) | 1B | 2 | R1,R2 |
| SRA | Shift right single | 8A | 4 | R1,D2(B2) |
| SRDA | Shift right double | 8E | 4 | R1,D2(B2) |
| SRDL | Shift right double logical | 8C | 4 | R1,D2(B2) |
| SRL | Shift right single logical | 88 | 4 | R1,D2(B2) |
| SRP | Shift and round decimal | F0 | 6 | D1(L1,B1),D2(B2),I3 |
| * SSK | Set storage key | 08 | 2 | R1,R2 |
| * SSM | Store shaded memory | DA | 6 | D1(L,B1),D2(B2) |
| * SSP | Store scratchpad | D0 | 6 | D1(L,B1),D2(B2) |
| ST | Store (word) | 50 | 4 | R1,D2(X2,B2) |
| 1)STBF | Store bit field | 81 | 4 | B1(D1),D3(L3) |
| STC | Store character | 42 | 4 | R1,D2(X2,B2) |
| STCK | Store clock | B2 | 4 | D1(B1) |
| STCM | Store character under mask | BE | 4 | R,M,D(B) |
| STD | Store, long | 60 | 4 | R1,D2(X2,B2) |
| STE | Store, short | 70 | 4 | R1,D2(X2,B2) |
| STH | Store halfword | 40 | 4 | R1,D2(X2,B2) |
| STM | Store multiple | 90 | 4 | R1,R3,D2(B2) |
| 1)STWI | Store word indirect | 51 | 4 | R1,D2(X2,B2) |

A

**Instruction format (SET1)**

| Mnemonic Code | Instruction Name | Mach. Code | Length | Operand Format |
|---|---|---|---|---|
| SU | Subtract unnormalized, short | 7F | 4 | R1,D2(X2,B2) |
| SUR | Subtract unnormalized, short (register) | 3F | 2 | R1,R2 |
| SVC | Supervisor call | 0A | 2 | I |
| SW | Subtract unnormalized, long | 6F | 4 | R1,D2(X2,B2) |
| SWR | Subtract unnormalized, long (register) | 2F | 2 | R1,R2 |
| SXR | Subtract normalized, extended | 37 | 2 | R1,R2 |
| * TDV | Test device | 9D | 4 | D1(B1) |
| TM | Test under mask | 91 | 4 | D1(B1),I2 |
| TR | Translate | DC | 6 | D1(L,B1),D2(B2) |
| TRT | Translate and test | DD | 6 | D1(L,B1),D2(B2) |
| TS | Test and set | 93 | 4 | D1(B1) |
| UNPK | Unpack | F3 | 6 | D1(L1,B1),D2(L2,B2) |
| * WRD | Write direct | 84 | 4 | D1(B1),I2 |
| X | Exclusive OR (word) | 57 | 4 | R1,D2(X2,B2) |
| XC | Exclusive OR (characters) | D7 | 6 | D1(L,B1),D2(B2) |
| XI | Exclusive OR immediate | 97 | 4 | D1(B1),I2 |
| XR | Exclusive OR (register) | 17 | 2 | R1,R2 |
| ZAP | Zero and add decimal | F8 | 6 | D1(L1,B1),D2(L2,B2) |

* Privileged instruction

1) Incompatible instruction

2) Hardware upgrading required

## A.7 INSTRUCTION FORMAT (SET2)

Additional or alternative instructions for the central processing units in the 7800 series.
Generation can be activated via the COMOPT INSTR=SET2 (see section 6.1.1.).

| Mnemonic Code | Instruction Name | Mach. Code | Length | Operand Format |
|---|---|---|---|---|
| * CLRCH | Clear Channel | 9F01 | 4 | D2(B2) |
| * CLRIO | Clear I/O | 9D01 | 4 | D2(B2) |
| * CONCS | Connect Channel Set | B200 | 4 | D2(B2) |
| * DISCS | Disconnect Channel Set | B201 | 4 | D2(B2) |
| EPAR | Extract primary ASN | B226 | 4 | R1 |
| ESAR | Extract secondary ASN | B227 | 4 | R1 |
| *1)HDV | Halt Device | 9E01 | 4 | D2(B2) |
| * HIO | Halt I/O | 9E00 | 4 | D2(B2) |
| IAC | Insert Address Space Control | B224 | 4 | R1 |
| IPK | Insert PSW Key | B20B | 4 | no operand |
| * IPTE | Invalidate Page Table Entry | B221 | 4 | R1,R2 |
| *1)ISK | Insert Storage Key | 09 | 2 | R1,R2 |
| * ISKE | Insert Storage Key extended | B229 | 4 | R1,R2 |
| IVSK | Insert virtual Storage Key | B223 | 4 | R1,R2 |
| * LASP | Load Address Space Parameters | E500 | 6 | D1(B1),D2(B2) |
| *2)LCTL | Load Control | B7 | 4 | R1,R3,D2(B2) |
| *2)LPSW | Load PSW | 82 | 4 | D2(B2) |
| *2)LRA | Load real Address | B1 | 4 | R1,D2(X2,B2) |
| 2)MVCP | Move to primary | DA | 6 | D1(R1,B1),D2(B2),R3 |
| MVCS | Move to secondary | DB | 6 | D1(R1,B1),D2(B2),R3 |
| 1)PC | Program Call | B218 | 4 | D2(B2) |
| PT | Program Transfer | B228 | 4 | R1,R2 |
| * PTLB | Purge TLB | B20D | 4 | no operand |
| * 3)RDD | Read direct | 85 | 4 | D1(B1),I2 |
| * RIO | Resume I/O | 9C02 | 4 | D2(B2) |
| * RRB | Reset Reference Bit | B213 | 4 | D2(B2) |
| * RRBE | Reset Reference Bit extended | B22A | 4 | R1,R2 |
| SAC | Set Address space control | B219 | 4 | D2(B2) |
| * SCK | Set Clock | B204 | 4 | D2(B2) |
| * SCKC | Set Clock Comparator | B206 | 4 | D2(B2) |
| * SIGP | Signal Processor | AE | 4 | R1,R3,D2(B2) |
| * SIO | Start I/O | 9C00 | 4 | D2(B2) |
| * SIOF | Start I/O fast release | 9C01 | 4 | D2(B2) |
| SPKA | Set PSW Key from Address | B20A | 4 | D2(B2) |
| * SPT | Set CPU Timer | B208 | 4 | D2(B2) |
| * SPX | Set Prefix | B210 | 4 | D2(B2) |
| SSAR | Set secondary ASN | B225 | 4 | R1 |
| *3)SSK | Set Storage Key | 08 | 2 | R1,R2 |
| * SSKE | Set Storage Key Extended | B22b | 4 | R1,R2 |
| *1,2)SSM | Set System Mask | 80 | 4 | D2(B2) |
| * STAP | Store CPU Address | B212 | 4 | D2(B2) |
| * STCK | Store Clock | 205 | 4 | D2(B2) |
| * STCKC | Store Clock Comparator | B207 | 4 | D2(B2) |
| *2)STCTL | Store Control | B6 | 4 | R1,R3,D2(B2) |
| * STIDC | Store Channel ID | B203 | 4 | D2(B2) |
| * STIDP | Store CPU ID | B202 | 4 | D2(B2) |
| *2)STNSM | Store Then and System Mask | AC | 4 | D1(B1),I2 |
| *2)STOSM | Store Then or System Mask | AD | 4 | D1(B1),I2 |

A

**Instruction format (SET2)**

| Mnemonic Code | Instruction Name | Mach. Code | Length | Operand Format |
|---|---|---|---|---|
| * STPT | Store CPU Timer | B209 | 4 | D2(B2) |
| * STPX | Store Prefix | B211 | 4 | D2(B2) |
| * TB | Test Block | B22C | 4 | R1,R2 |
| * TCH | Test Channel | 9F00 | 4 | D2(B2) |
| * TIO | Test I/O | 9D00 | 4 | D2(B2) |
| * TPROT | Test Protection | E501 | 6 | D1(B1),D2(B2) |
| *3)WRD | Write direct | 84 | 4 | D1(B1),I2 |

* Privileged instruction

1) Same mnemonic code as for instruction set SET1

2) Same machine code as for instruction set SET1

3) Identical with the instruction with the same name in instruction set SET1

# A.8 INTERNAL ASSEMBLY SOURCE LANGUAGE UPDATE ROUTINE (ISLU)

## 1. Introduction

The Internal Assembly Source Language Update routine (ISLU) provides the user with the capability to store and maintain assembly language source programs on magnetic tape and disk.

Depending on the options chosen, the source programs may be retrieved from source library tapes or disk files. An output library tape with standard tape labels can be created by this routine, or source programs can be stored on disk files (i.e., ISAM or SAM).

## 2. Input

### Source Corrections

Source corrections may be entered from a disk file, a terminal, or a card reader.

If the corrections are entered on a terminal or retrieved from a disk file, it is not necessary for the records to be 80-column card images. If the last character in the input record is not a space, an 8-byte sequence number is assumed to be present. The input record is reformatted so that the sequence number is in columns 73 to 80, and spaces are inserted between the end of the remaining data and the sequence number. If the last character of the input record is a space, the record is expanded to 80 bytes by adding spaces to the right-hand end.

### Source Library File

A source library tape may have either standard or nonstandard tape labels with the linkname equal to BINPUT. However, this routine allows only standard labels for the library output tape. BOUTPUT is the output linkname.

The source library tape may contain a single program or multiple programs in any order, but is confined to a single reel. Each program consists of a number of blocks containing from one to five 80-column source statement images, preceded by an 80-column *STARTC image and followed by a tape mark. The last program on the tape may be followed by either a standard or nonstandard EOF tape label.

The source file may also reside on a disk as a SAM or ISAM file. All records are V-type records and ISAM records must contain an 8-byte key field in the first data position. ISAM keys are not considered to be part of the source statement, and they are ignored by this source language update facility.

Source statements do not have to be 80-column card images. If source statements are less than 80 bytes (i.e., record length less than 84 for SAM and less than 92 for ISAM files), the source will be expanded to 80 bytes by adding blanks on the right. If the source statement is longer than 80 bytes, it will be truncated on the right. When sequence numbers are used, however, they must be in columns 73 to 80.

The linknames for disk input and disk output files are DINPUT and DOUTPUT.

## 3. Output

### Source Library Tape

This tape has the following format

1. Standard tape labels
2. 80-character *STARTC program identifier block
3. Source statement blocks, 5 statements per block
4. Tape mark separating each program
5. Standard label processing at CLOSE

### Disk Output Files

A source program may be stored on disk as an ISAM or SAM file.
The disk files have the following format:

1. All records are V-type records (84 bytes for SAM and 92 bytes for ISAM)
2. Only one source program per cataloged file.
3. ISAM keys starting at 100 with increments of 100. (Note: ISAM keys are not connected with the sequence numbers in columns 73 to 80 in any way.)

### Correction Listing

Corrections made to source programs are always listed. Whenever a source statement is replaced, the new statement and the first 38 bytes of the old statement are listed. All the deleted old statements are listed also.

## 4. Control Statements for ISLU

Control statements accepted by the Internal Assembly Source Language Update Facility are:

STARTC, DELETE, ENDC

### *STARTC Statement

When the source library update facility is desired, a *STARTC control statement must be the first statement read from the SYSDTA file. This statement can be continued once. Columns 1 through 7 must contain *STARTC.

| Operation | Operand |
|---|---|
| *STARTC | progname,[option],[SEQ],[number],[size],[id]<br>[,INPUT=filename][,OUTPUT=filename] |

All positional operands except progname are optional. A comma must be used to denote a missing operand unless no more positional operands follow. The two key-word operands, if present, must appear after the positional operands. A hyphen (-) is used for continuation provided that it always appears after the positional operands and a comma, of after a comma followed by spaces.

The continuation statement can start from any column. The program named in the *STARTC statement is always assembled, and the *STARTC statement from the SYSDTA file always replaces the *STARTC statement on the output type or disk file.

progname — The program name must be preceded by at least one space and can be any combination of characters except space and comma. Maximum length is eight characters.

option — This operand may be one of six options as follows:

1. Unspecified: The source program, which must be in the SYSDTA file, is assembled and written to the output tape or disk. This option is used for initial creation of the source library tape or disk file. The unspecified option is indicated on the *STARTC statement as a null operand.

2. ADD: All programs on the input tape are copied to the output tape, then the program to be added, which must be in the SYSDTA file is assembled and written to the output tape. Corrections may not be applied. If more than one program is to be added, the succeeding programs must use option 1. This option is not allowed if any keyword operand is specified.

3. ASSEMBLE: The choice of this option permits the specified program on the input tape or disk to be assembled with no corrections. Neither an output tape nor a disk file is created and the *EMDC statement must be omitted.

4. CORRECT: The source program from the input tape or disk is updated with corrections from the SYSDTA file and assembled. No output files are created.

5. COPY: The source program from the input tape or disk is updated with corrections from the SYSDTA file, assembled, and written to the output tape or disk. No other programs from the input tape are processed.

6. COYPALL: This function is identical to COPY, except that (1) all programs on the source library input prior to the one to be assembled/corrected are first copied to the source output and (2) this option is not allowed if any keyword operand is used.

SEQ — Optional, if present, it instructs the assembler to insert sequence numbers in the updated source program.
If this operand is not specified, the contents of columns 73 through 80 of the source statements or correction statements are retained.

number — Optional and should be used only in conjunction with the SEQ operand. If SEQ is not specified, the number operand is ignored when present. The number operand specifies the first sequence number to be assigned in a resequenced output file. If SEQ is specified and no number operand is specified, then 100 is assumed. Sequence numbers are incremented by 100 in all cases.

A

size — This optional field specifies the number of digits in the sequence number and must be a value between four and eight. For example, if 4 is specified, the sequence number is placed in columns 77 through 80. If the field is not specified an eight-character field (columns 73 through 80) is assumed. If the number of digits specified in the number field exceeds that specified by the size filed or the implied size field, the rightmost digits of the number field are used.

id — This operand is ignored if the SEQ operand is omitted. This operand specifies an identification field that will be inserted into all source statements beginning in column 73. If SEFQ is used, the id field length is the difference between the maximum (8) less the number of characters specified in the size operand.

INPUT=filename
This option is used only if the source program is retrieved from disk, and filename is the cataloged file which contains the source program (i.e., ISAM or SAM) on disk. The linkname for disk input files is DINPUT:

OUTPUT=filename
This option must be specified if the assembled and corrected source program is to be created as a disk file (i.e., ISAM or SAM); filename refers to the file which will contain the source program. The linkname for disk output is DOUTPUT.

***DELETE Statement**

Whenever deletion of one or more cards is desired, a *DELETE statement is required. *DELETE appears in column 1 through 7 and an optional comma may appear in column 8.

| Operation | Operand |
|---|---|
| *DELETE | [,]d1[,d2] |

d1 — Specifies the sequence number of the first card to be deleted and begins in column 9.

d2 — Specifies the sequence number of the last card to be deleted and must be equal to or greater than d1.

Progamming Notes:

1. d1 and d2 are any combination of characters except space or comma. If the field is greater than eight characters, the rightmost eight characters are used. If less than 8 characters, the sequence field is right-justified and space-filled to the left. If d2 is omitted, then it is set equal do d1.

2. The comma in column 8 is optional. If present, the next eight characters are considered as the d1 operand, regardless of their value. This option allows deletion of individual statements which contain invalid characters in the sequence number field. In order to properly position the source tape, a "dummy" correction should be given to the preceding statement containing a valid sequence number.

### *ENDC Statement

The final control statement of all programs being corrected must be the *ENDC statement unless the ASSEMBLE option is used. If corrections are present, this statement must follow the last correction statement. If no corrections are present, it must follow the *STARTC statement. *ENDC appears in column 1 through 5.

| Operation | Operand |
|---|---|
| *ENDC | [COPY] |

COPY Directs the assembler to copy the remaining programs on the source library input tape to the updated source output tape. This operand is optional.

Programming Note:

If the *STARTC statement specifies the CORRECT option, and the COPY operand is present in the *ENDC statement, the COPY operand is rejected.

A

### 5. Correction Statements

Correction statements are identified by exception: that is, if a statement does not begin with *STARTC, *DELETE, or *ENDC, it is processed as a correction. Correction statements must be in the SYSDTA file in ascending order by sequence number (columns 73 through 80).

Correction statements are divided into two categories:

**Replacement and insertion.** If the sequence field of a correction statement is equal to the sequence number of a source library statement, then the source library statement is replaced by the correction statement. If a correction statement has a sequence number that is not equal to the sequence number of any statement on the tape, then the statement is inserted in proper numerical order. If a correction statement has a space in column 80, it is considered to be an insertion and is inserted immediately. Thus, by utilizing dummy replacements or insertions to position the input tape, large sections of new coding may be inserted.

## 6. ISLU Messages

| Message | Meaning | Response |
|---|---|---|
| *ERROR* | 1. d1 greater than d2 in *DELETE statement. | d2 is set equal do d1. |
| | 2. Option operand in *STARTC statement invalid | Blank operand assumed. Rest of statement is ignored. |
| *ERROR* FATAL | Program to be corrected cannot be located on source library input tape. | If COPYALL is used in STARTC, the source input is copied to source output. |
| NO*ENDC | 1. No *ENDC statement to terminate statements.<br>2. Correction statements out of sequence. | Correct and restart. |

## 7. Examples:

1. To use the unspecified option to produce a source library tape for a program called PROGA from the SYSDTA file.

```
*STARTC PROGRA,,SEQ
  .
  .                    } Source input for PROGA
  .
*ENDC
```

2. To use the unspecified option to create a disk file, SAM.FILE.NAME, for source program called PROGB from the SYSDTA file.

```
*STARTC PROGB,,SEQ,OUTPUT=SAM.FILE.NAME
  .
  .                    } Source input for PROGB
  .
*ENDC
```

3. A specific source program called PROGC on input tape to be assembled only and another source program called PROGD on the same input tape to be assembled and corrected without creating an output tape or disk file.

```
*STARTC PROGC,ASSEMBLE
*STARTC PROGD,CORRECT
  .
  .                    } Source statements to update PROGD
  .
*ENDC
```

4. A disk file whose name is ASM.PROGA.SAM to be assembled and updated, then stored back to disk as ASM.PROGA.UPDATE.

```
*STARTC PROGA,COPY,INPUT=ASM.PROGA.SAM,-
OUTPUT=ASM.PROGA.UPDATE
  .                                  } Source statements to update PROGA
  .
  .
*ENDC
```

5. A disk file whose name is ASM.PROGA.INPUT to be assembled, updated and stored on disk as an ISAM file named ISAM.PROGA.OUTPUT.
   Because the output file is an ISAM file, the user must issue a FILE command for the output file.

A

## A.9 ASSEMBLER RESTRICTIONS

The F-Assembler cannot translate arbitrarily large programs. The size of a program, the number of symbols, and other limitations must be observed to assure a correct translation. The limitations imposed by the assembler are detailed below.

### 1. Statement Numbers in the Assembly Listing

In the assembly listing every statement is given a statement number. An unlimited number of statements can be translated, but the statement numbers are assigned modulo 99999.

### 2. Size of Diagnostic File

A diagnostic file is created when PARAM SAVLIST=ALL or *COMOPT SAVLST is specified. This file has a capacity of 98303 statements. If it is attempted to enter more than this number of statements in the file, the F-Assembler issues the error message DIAGNOSTIC FILE OVERFLOW - FILE CLOSED and the diagnostic file is closed.



### 3. Entries in XREF Listing

The XREF Listing is limited to 1 000 000 entries. An entry is the number of a statement in which a symbol or a macro is defined or invoked. If a XREF Listing is requested when the number of entries exceeds 1 000 000, the error message XREF TABLE OVERFLOW: XREF AND DIAGNOSTIC LISTING INCOMPLETE is issued and only the first 1 000 000 references are printed.

### 4. Maximum Number of Symbols

The maximum number of symbols and macro names depends on the available class-6 storage:

- with 1MB user address space: 10580;
- with 2MB user address space: 34900;
- with user address space >2MB: 43000.

If more than this number are used, the assembler issues the error message 'SYMBOL TABLE OVERFLOW - ASSEMBLY TERMINATED' and terminates the translation.

### 5. Maximum Number of ESID Numbers

The assembler can assign a maximum of 2500 ESID numbers. The following items are assigned individual ESID numbers:

- program sections (Type SD)
- common control sections (Type CM)
- dummy sections (Type DS)
- external dummy section (Type XD, XR)
- XDSEC symbols (Type XD, XR)
- EXTRN symbols (Type ER)
- WEAK-EXTRN symbols (Type WX)
- V constants (Type VC)
- dummy registers (Type DX)

### 6. Maximum Number of Macro Definitions in the Macro Library

The macro libraries used by the F-Assembler may contain up to 104000 macro definitions.

### 7. Maximum Number of macro names in MCALL Statements

A maximum of 400 macro names can be used in the MCALL statements.

### 8. MCALL Control

A macro library may comprise macro definitions containing inner macro instructions, where these macros are also contained in the library.

If process switches 0 and 31 are both set, then, in the case of inner macro instructions, only those macros are present in the library for the F-Assembler whose macro names are explicitly specified in MCALL statements in the source program. Only such inner macro instructions can be expanded.

### 9. Nesting Depth of Macros

The maximum nesting depth of the macros is 255. In the assembly listing however, only two columns are provided for this. The nesting depth is therefore indicated modulo 100.

### 10. Default Values

Before SET symbols are used, they must be defined.

Excepted from this requirement are the default values. These can be assigned values without first being defined.

Default values are SET symbols with special names:

- Global and local SETA symbols having the forms &AGm and &ALm respectively, where m is a decimal number between 0 and 99.
- Global and local SETB symbols having the form &BGn and &BLn respectively, where n is a decimal number between 0 and 999.
- SETC symbols are exclusively global and have the form &CGm, where m is a decimal number between 0 and 99.

### 11. Length of Input Records

The assembler accepts input records of lengths up to 128 characters. However, only the first 80 bytes are used and further processed, the rest of the record has no significance for the F-Assembler. If an input record contains more than 128 characters, the error message

'READ TRUNCATION ERROR IN RDATA MACRO'

and then the input record in question are output. The translation is then discontinued.

## 12. Operand Length in Macro Instructions

In Section 4.13.1 it is specified that the maximum length of individual operands in macro instructions is 127 characters. However, this is only an approximate value. The exact value of the character length of an operand can be derived from the internal byte length. The maximum internal length for an operand is 240 bytes.

An operand in a macro instruction can be a

- SYSNDX symbol
- SYSLIST symbol
- symbolic parameter
- symbolic operand
- arithmetic operand
- character operand
- composite operand
- sublist
- label operand.

**a) SYSNDX symbol**

**Internal length:** 3 bytes

Example: operand &SYSNDX
length 3 bytes

**b) SYSLIST symbol**

Internal length: 7 bytes

Example: operand &SYSLIST (3)
length 7 bytes

**c) Symbolic parameter**

Internal length: 3 bytes

Example: operand &AAAA
length 3 bytes

**d) Symbolic operand**

Form: The operand consists of up to 8 characters, the first character being a letter, $, #, or @.

Internal length: 5 bytes + length of symbol

Example: operand BBBB
length 9 bytes.

**e) Arithmetic operand**

Form: The operand is a decimal number of up to 8 digits.

Internal length: 5-15 bytes, depending on the size of the decimal number.

Example: operand 3333
length 9 bytes.

**f) Character operand**

Form: The operand is a character string other than d) or e) (character string > 8 characters).

Internal length: 6 bytes + length of character string

Note: The operand may contain up to 234 characters.

Example:

| | |
|---|---|
| operand | CCCCCCCCCC |
| length | 16 bytes. |

**g) Composite operand**

Format:

| | |
|---|---|
| 4 bytes | Suboperands |

An operand can be made up of suboperands of the types a), b), c), d), e) and f). The composite operand is treated by the F-Assembler as a character operand.

Note: The sum of the internal length of the suboperands must not exceed 236 bytes.

Example:

| | |
|---|---|
| operand | AAAA&BB |
| length | 16 bytes |

**h) Sublist**

Format:

| | |
|---|---|
| 5 bytes | Sublist parameters |

The sublist parameters can be of type a), b), c), d), e), f), g) or h).

Note: The sum of the internal lengths of the sublist parameters must not exceed 235 bytes.

Example:

| | |
|---|---|
| operand | (AAAA,BBBB,CCCC) |
| length | 32 bytes |

**i) Keyword operand**

Format:

| | | |
|---|---|---|
| 2 bytes | Keyword | Keyword parameter |

Keyword parameters can be of type a), b), c), d), e), f), g) or h).

Note: The sum of the internal lengths of the keyword parameters must not exceed 238 bytes - keyword length.

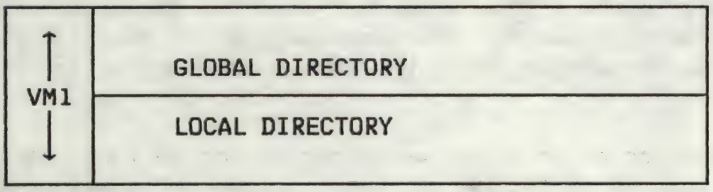
Example:

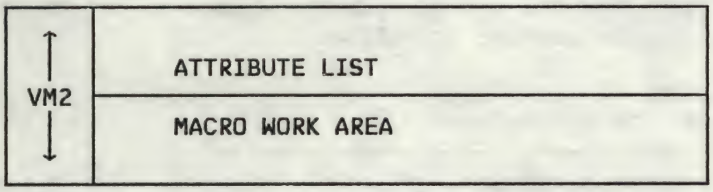
| | |
|---|---|
| operand | X=AAAA |
| length | 12 bytes |

## 13. Global and local Variables

The maximum number of global and local variables that can be used cannot be exactly specified because of the large number of factors involved in an estimate. Roughly approximate values are given below:

The work areas, VM1 and VM2, are provided for global and local variables.




A

Starting with Assembler Version 29.0 the sizes of the VM1 and VM2 areas depend on the size of class-6 storage and whether the assembler was implemented in shareable mode or not.
The following table summarizes the possible sizes:

| | Assembler | | | Assembler shared | | |
|---|---|---|---|---|---|---|
| | Class-6 memory | | | Class-6 memory | | |
| | 1MB | 2MB | > 2MB | 1MB | 2MB | > 2MB |
| VM1 | 40 KB | 120 KB | 120 KB | 40 KB | 120 KB | 120 KB |
| VM2 | 48 KB | 136 KB | 264 KB | 168 KB | 136 KB | 264 KB |

In the **GLOBAL DIRECTORY** are stored

- global SET symbols
- local SET symbols defined in the source program
- sequence symbols defined in the source program
- GSEQ symbols defined in source program

In the **LOCAL DIRECTORY** are stored

- symbolic parameter
- SYSLIST symbols
- local SET symbols defined in macros
- sequence symbols defined in macros
- GSEQ symbols defined in macros.

In the **Attribute List** are stored:

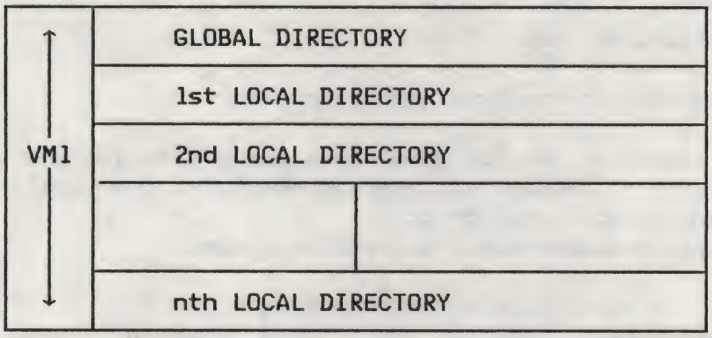
- symbols in the source program whose attributes are accessed
- operands of outer macro instructions.

In the **Macro Work Area** are stored

- current values of global SETC symbols.

The GLOBAL DIRECTORY belongs to the source program, a LOCAL DIRECTORY is assigned to every macro definition. The GLOBAL DIRECTORY, Attribute List and macro work area are therefore located in the MV1 or MV2 areas during the entire translation of the source program, but the LOCAL DIRECTORY is only attached to the GLOBAL DIRECTORY when the associated macro instruction is being expanded. This storage area is thereafter not required and remains unused until the next macro instruction is processed.

During the expansion of an outer macro containing an inner macro instruction, the LOCAL DIRECTORY of the inner macro is attached to that of the outer macro for the processing of the inner macro. If the macro nesting depth is n, the VM1 area during the expansion of the n-th level inner macro can be represented as follows:




The consequences of this organization are as follows:

**a) General**

The areas for the GLOBAL DIRECTORY, LOCAL DIRECTORIES, and the Attribute List cannot exceed 32 Kbytes, the macro work area is limited to 64 Kbytes.

**b) For the VM1 Area**

**Problem:**

If a large number of local variables are used, particularly LCLC symbols with frequently changing values, and if a high degree of nesting occurs in the macros, it is possible that the VM1 area overflows. The assembler then issues the error message 'Z3-LOCAL DICTIONARY OVERFLOW'.

**Remedies:**

There are two possibilities: The user can reduce the number of local variables, particularly of the LCLC symbols with frequently changing values. With some local variables it may be possible to shift to the global area. In the case of SETC symbols, more storage space is available in the global area than in the local area. The other possibility is to reduce the depth of nesting of the macros.

**c) For the VM2 Area**

**Problem:**

The current values of global SETC symbols are stored in the VM2 area. If a large number of GBLC symbols with frequently changing current values are used, it is possible that the VM2 area overflows. The assembler then issues the error message 'Z2-GLOBAL DICTIONARY OVERFLOW'.

**Remedy:**

The user must reduce the number of GBLC symbols. In the case of some symbols it may be possible to shift to the local area.

## A.10 HALSTEAD METRICS

Complexity metrics of source programs can be output by the assembler.

The definitions and measurements used are based on a theory developed by Maurice H. Halstead.

The major measurements are:

Program volume, most compact program volume, program level and programming effort.

These measurements are computed from four metrics:

- Number of distinct operators
- Number of distinct operands
- Total number of operators
- Total number of operands.

This facility can be activated by COMOPT control using the PROCOM option (default: NOPROCOM).

**Description of program complexity metrics:**

In the nonexpanded program the assembler counts

m1 = the number of distinct operators

m2 = the number of distinct operands

n1 = the total number of operators

n2 = the total number of operands

From this we compute:

m: = m1 + m2: The number of distinct program operators and operands, the program vocabulary.

n: = n1 + n2: The total number of operators and operands, the program length.

v: = n * log2 (m):
The program volume.
In order to distinguish between m distinct operators and operands, at least log2 (m) bits storage are required per element. Thus v is the minimum number of bits to be allocated to store the program.

v* = 2 (const.): The most compact program volume.
The smallest possible implementation of any assembly program consists of a parameterless macro instruction.

l: = v*/v: The program level (0<l<=1).
The program level is defined as the ratio of the most compact to the actual volume. The highest level is 1 (the level is 1 if there is a macro that corresponds to the program). The program level decreases as the actual program volume increases.

e: = v/l: The programming effort.
e is the ratio of program volume and program level.
e increases as the volume increases and the level decreases; and e decreases in the other case.

**Definition of operators and operands**

Operators and operands are counted in the unexpanded source program.

All keywords of the assembler are regarded as "operators". These are:

- All machine instructions
- All assembly statements (except for the TITLE, STACK, UNSTK, EJECT, SPACE; PRINT, ICTL and ISEQ statements, which merely control the assembly listing or the I/O).
- The special characters, () + - * / = '
- The attributes I, K, L, N, S, T
- In constants, literals and direct data: the constant types: C, X, B, F, H, E, D, L, P, Z, A, Y, S, V and Q and the modifiers L, S and E
- Each end of statement
- The boolean operators NOT, AND, OR, EQ, NE, LE, LT, GE and GT.

**Operands** are all user-defined lexical elements. These are:

- Symbolic names, symbolic parameters and sequence symbols
- Constants, literals, self-defining values and character strings. If apostrophes or parentheses are used, everything that is enclosed will be considered as an operand unless several values are separated by commas.
- Macro names, macro operands and keywords
- Duplication factors, lengths and modifiers
- Masks
- Registers

Only the lexical elements are counted.

So if, for example, the number 5 occurs as an operand in the source program, then no distinction is made whether this is a length, a register or a constant.

If in one position there is a duplication factor 5 and in another the register indication 5, both numbers will achieve an increase of m2 by 1, and an increase of n2 by 2.

Also for operators, e.g. for L, no distinction is made between constant type and length attribute. When a count is made, comment lines and notes will not be considered.

As a result of the measurements, the program complexity metrics are printed at the end of the listing, in the following form:

```
UNIQUE OPERATORS    M1:    n
UNIQUE OPERANDS     M2:    n
USAGE OPERATORS     N1:    n
USAGE OPERANDS      N2:    n
VOCABULARY          M :    n
OBSERVED LENGTH     N :    n
PROGRAM VOLUME      V :    n
PROGRAM LEVEL       L :    n
PROGRAMMING EFFORT  E :    n
```

## A.11 EXAMPLES OF DUMMY REGISTERS

Comments explain the functions of the programs so that further explanations need not be given. Note that the method used in Example 1 is appropriate for small programs consisting of few CSECTS; for larger programs the dummy register vector may be allocated using REQM. Surely, this does not exclude the use of Q-type constants and a base register for dummy register addressing. In Example 2, the dummy register vector starts at address X'1000', the first page allocated for the dummy register vector.

```
DUMMY REGISTER VECTOR EXAMPLE 1 CSECT1                                                    16:20:42  86-09-01   PAGE 0002

FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT
                                          1              TITLE 'DUMMY REGISTER VECTOR EXAMPLE 1 CSECT1'
                                          2     *
                                          3     *            - IN THIS EXAMPLE THE LENGTH OF THE DUMMY
                                          4     *              REGISTER VECTOR IS DEFINED AND STORED
                                          5     *              AT PROGRAMMING TIME.
                                          6     *            - THE INDIVIDUAL DUMMY REGISTERS ARE
                                          7     *              ADDRESSED BY MEANS OF Q-TYPE CONSTANTS.
                                          8     *
                                          9              PRINT NOGEN
   000000                                10     PSEUDO1  CSECT
                                         11              EXTRN PSEUDO2
   0000FC                                12              ENTRY PSRVEKT
   000000 05 A0                          13              BALR  10,0
   000002                                14              USING *,10
                                         15     *
   000002 41 10 A0FA     0000FC          16              LA    1,PSRVEKT
   000006 58 20 A0EE     0000F0          17              L     2,QPSREG1          R2 < - DISPLACEMENT OF DUMMY REGISTER 1
   00000A 1A 21                          18              AR    2,1              RELATIVE TO START OF DUMMY REGISTER VECTOR
   00000C D2 09 2000A19A        00019C   19              MVC   0(10,2),=C'WUNDERLICH'      TO BECOME CONTENTS OF PSREG 1
   000012 58 20 A0F6     0000F8          20              L     2,QPSREG3          R2 < - DISPLACEMENT OF DUMMY REGISTER 3
   000016 1A 21                          21              AR    2,1              RELATIVE TO START OF DUMMY REGISTER VECTOR
   000018 D2 4F 2000A09C        00009E   22              MVC   0(80,2),TEXT            'TEXT' TO BECOME CONTENTS OF PSREG 3
   00001E 58 E0 A196     000198          23              L     14,=A(PSEUDO2)
   000022 05 FE                          24              BALR  15,14
   000024                                25              WROUT NACHR1,FEHLER
                                         39  2                 *,ADCEO      800    831115   53531004                    00063000
                                         41  1                 *,WROUT      800    831216   53121058
   000032                                42              TERM
   00004A                                54     FEHLER   TERM  DUMP=Y
                                         66     *
   000064                                67     NACHR1   DS    0F
   000064 0038000040                     68              DC    X'0038000040'            X'38' = MESSAGE LENGTH
   000069 5C5C5C5C5C5C5C5C               69              DC    C'************* PROGRAMMENDE BEISPIEL 1 ***************'
   00009E C5C9D540D4C5D5E2               70     TEXT     DC    C'EIN MENSCH KANNS MANCHMAL NICHT VERSTEHN,'
   0000C7 E3D9C9C6C6E340C5               71              DC    C'TRIFFT EIN WAS ER VORAUSGESEHN.          '
                                         72     *
   000000                                73     PSREG1   DXD   CL10                    DEFINITIONS OF DUMMY REGISTERS 1,
   000000                                74     PSREG2   DXD   CL60                                            2 AND
   000000                                75     PSREG3   DXD   20F                                             3
   0000F0 00000000                       76     QPSREG1  DC    Q(PSREG1)                DEFINITIONS OF Q-TYPE CONSTANTS OF
   0000F4 00000000                       77     QPSREG2  DC    Q(PSREG2)          DUMMY REGISTER VECTORS FOR ADDRESSING THE
   0000F8 00000000                       78     QPSREG3  DC    Q(PSREG3)                   INDIVIDUAL DUMMY REGISTERS
   0000FC                                79     PSRVEKT  DS    CL(L'PSREG1+L'PSREG2+L'PSREG3)
   000198 00000000                       81                    =A(PSEUDO2)
   00019C E6E4D5C4C5D9D3C9               82                    =C'WUNDERLICH'
   000000                                83              END   PSEUDO1
```

```
FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT
                                           1            TITLE 'DUMMY REGISTER VECTOR EXAMPLE 1 CSECT2'
                                           2            PRINT NOGEN
   000000                                  3    PSEUDO2 CSECT
   000000 05 B0                            4            BALR  11,0
   000002                                  5            USING *,11
                                           6            EXTRN PSRVEKT
   000002 58 10 B016     000018            7            L     1,=A(PSRVEKT)
   000006 58 20 B012     000014            8            L     2,QPSREG2          R2 < - DISPLACEMENT OF DUMMY REGISTER 2
   00000A 1A 21                            9            AR    2,1                RELATIVE TO START OF DUMMY REGISTER VECTOR
   00000C D2 13 2000B01A        00001C    10            MVC   0(20,2),=C'(E. ROTH          )'
   000012 07 FF                           11            BR    15
                                          12    *
   000000                                 13    PSREG2  DXD   CL22
   000014 00000000                        14    QPSREG2 DC    Q(PSREG2)
   000018 00000000                        16                  =A(PSRVEKT)
   00001C 4DC54B40D9D6E3C8                17                  =C'(E. ROTH
   000000                                 18            END   PSEUDO2
```

```
(IN)     LOAD PSEUDO1
(OUT)    %  P500 PSEUDO1/000/02-86-09 LOADED
(OUT)    (MSG) %  %  E560 PSEUDO1
(IN)     AT L'32'
(IN)     R
(OUT)    ************* PROGRAMMENDE BEISPIEL 1 *************
(OUT)    INTERRUPTED AT 000032
(IN)     D L'0':L'10000'
(OUT)     VM-ADR 000000:010000  P-COUNT 000032
(OUT)     000000   05A04110  A0FA5820  A0EE1A21  D2092000  A19A5820 ............K.......
(OUT)     000014   A0F61A21  D24F2000  A09C58E0  A19605FE  4510A02E .6..K|.......o......
(OUT)     000028   00000064  0000004C  0A430A5B  0A1C0700  4510A046 .......<...$........
(OUT)     00003C   01000004  40404040  00000075  0A090700  0A1C0700 ....    ............
(OUT)     000050   4510A05E  01010004  40404040  00000075  0A090000 ...;....    ........
(OUT)     000064   00380000  405C5C5C  5C5C5C5C  5C5C5C5C  5C5C40D7 .... ************ P
(OUT)     000078   D9D6C7D9  C1D4D4C5  D5C4C540  C2C5C9E2  D7C9C5D3 ROGRAMMENDE BEISPIEL
(OUT)     00008C   40F1405C  5C5C5C5C  5C5C5C5C  5C5C5C5C  5C5CC5C9  1 ***************EI
(OUT)     0000A0   D540D4C5  D5E2C3C8  40D2C1D5  D5E240D4  C1D5C3C8 N MENSCH KANNS MANCH
(OUT)     0000B4   D4C1D340  D5C9C3C8  E340E5C5  D9E2E3C5  C8D56BE3 MAL NICHT VERSTEHN,T
(OUT)     0000C8   D9C9C6C6  E340C5C9  D540E6C1  E240C5D9  40E5D6D9 RIFFT EIN WAS ER VOR
(OUT)     0000DC   C1E4E2C7  C5E2C5C8  D54B4040  40404040  40400000 AUSGESEHN.        ..
(OUT)     0000F0   0000008C  00000000  0000003C  4DC54B40  D9D6E3C8 ............(E. ROTH
(OUT)     000104   40404040  40404040  4040405D  00000000  00000000            ).......
(OUT)     000118   00000000  00000000  00000000  00000000  00000000 ....................
(OUT)     00012C   00000000  00000000  00000000  C5C9D540  D4C5D5E2 ............EIN MENS
(OUT)     000140   C3C840D2  C1D5D5E2  40D4C1D5  C3C8D4C1  D340D5C9 CH KANNS MANCHMAL NI
(OUT)     000154   C3C8E340  E5C5D9E2  E3C5C8D5  6BE3D9C9  C6C6E340 CHT VERSTEHN,TRIFFT
(OUT)     000168   C5C9D540  E6C1E240  C5D940E5  D6D9C1E4  E2C7C5E2 EIN WAS ER VORAUSGES
(OUT)     00017C   C5C8D54B  40404040  40404040  E6E4D5C4  C5D9D3C9 EHN.        WUNDERLI
(OUT)     000190   C3C80000  00000000  000001A8  E6E4D5C4  C5D9D3C9 CH.........yWUNDERLI
(OUT)     0001A4   C3C80000  05B05810  B0165820  B0121A21  D2132000 CH..............K...
(OUT)     0001B8   B01A07FF  00000000  000000FC  4DC54B40  D9D6E3C8 ...|........(E. ROTH
(OUT)     0001CC   40404040  40404040  4040405D  00000000  00000000            ).......
(OUT)    =  VIRTUAL MEMORY ADDRESS 000FDC   THRU 010000   ARE NOT ALLOCATED
```

Program

Defined data constants

Dummy register vector

Literal pool

PSREG1

PSREG3

PSREG2

```
FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT
                                          1              TITLE 'DUMMY REGISTER EXAMPLE 2 CSECT1'
                                          2     *
                                          3     * - IN THIS EXAMPLE THE DUMMY REGISTER VECTOR IS STORED AT PROGRAM
                                          4     *   RUNTIME BY REQUESTING MEMORY PAGES. THE PROGRAMMER NEED NOT CONCERN
                                          5     *   HIMSELF WITH THE LENGTH OF THE DUMMY REGISTER VECTOR AS THIS IS ENTERED
                                          6     *   IN FIELD 'PRVLEN' (SEE INITIALIZATION ROUTINE) BY THE LINKAGE EDITOR.
                                          7     *   EVALUATION IS PERFORMED AT PROGRAM RUNTIME.
                                          8     * - ADDRESSING THE INDIVIDUAL DUMMY REGISTERS IS EFFECTED VIA A
                                          9     *   'DUMMY REGISTER VECTOR BASE REGISTER'.
                                         10     *
                                         11     *
                                         12             PRINT NOGEN
000000                                   13     PSEUDO3 CSECT
                                         14             EXTRN PRVINIT
000000 05 A0                             15             BALR  10,0
000002                                   16             USING *,10
000000                                   17             USING *PRV,8              DEFINES THE BASE REGISTER FOR ACCESS TO
                                         18     *                                                DUMMY REGISTER VECTOR
000002 58 E0 A0E6     0000E8             19             L     14,=A(PRVINIT)
000006 05 CE                             20             BALR  12,14
000008 18 81                             21             LR    8,1                  R8 < - A (1ST ALLOCATED PAGE)
                                         22     *
00000A D2 23 8000A0EA 000000 0000EC      23             MVC   PSREG1,=C'EIN MENSCH WIRD MUEDE SEINER FRAGEN:'
000010 D2 23 8000A07F 000000 000081      24             MVC   PSREG2,TEXT2                          MOVES DATA
000016 D2 20 8000A0A3 000000 0000A5      25             MVC   PSREG3,TEXT3                           TO DUMMY
00001C D2 1F 8000A0C4 000000 0000C6      26             MVC   PSREG4,TEXT4                          REGISTERS
000022                                   27             WROUT NACHR,FEHLER
                                         41  2                *,@DCEO      800    831115   53531004                    00063000
                                         43  1                *,WROUT      800    831216   53121058
000032                                   44     FEHLER  TERM
                                         56     *
00004C                                   57     NACHR   DS    0F
00004C 0035000040                        58             DC    X'0035000040'
000051 5C5C5C5C5C5C5C5C                  59             DC    C'************** PROGRAMMENDE BEISPIEL 2 *********'
000081 D5C9C540D2C1D5D5                  60     TEXT2   DC    C'NIE KANN DIE WELT IHM ANTWORT SAGEN.'
0000A5 C4D6C3C840C7C5D9                  61     TEXT3   DC    C'DOCH GERN GIBT AUSKUNFT ALLE WELT'
0000C6 C1E4C640C6D9C1C7                  62     TEXT4   DC    C'AUF FRAGEN,DIE ER NIE GESTELLT. '
                                         63     *
000000                                   64     PSREG1  DXD    CL36
000000                                   65     PSREG2  DXD    9F
000000                                   66     PSREG3  DXD    XL33
000000                                   67     PSREG4  DXD    4D
0000E8 00000000                          69                    =A(PRVINIT)
0000EC C5C9D540D4C5D5E2                  70                    =C'EIN MENSCH WIRD MUEDE SEINER FRAGEN:'
000000                                   71             END   PSEUDO3
```

DRV EXAMPLE 2 , INITIALIZATION 16:38:04 86-09-01 PAGE 0002

```
FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT

                                          1            TITLE 'PRV EXAMPLE 2 , INITIALIZATION'
                                          2            PRINT NOGEN
   000000                                 3     PRVINIT CSECT
   000000 05 B0                           4            BALR  11,0
   000002                                 5            USING *,11
                                          6     *
   000002 17 22                           7            XR    2,2          DEFINES NUMBER OF 4K PAGES REQUIRED
   000004 58 30 B03A     00003C           8            L     3,PRVLEN     IN ACCORDANCE WITH LENGTH OF
   000008 5D 20 B03E     000040           9            D     2,=F'4096'   DUMMY REGISTER VECTOR          AND...
   00000C 5A 30 B042     000044          10            A     3,=F'1'
   000010                                11            REQM  (3)          ...ALLOCATION OF IT.
   00001E 12 FF                          20            LTR   15,15
   000020 07 8C                          21            BRZ   12
   000022                                22            TERM  DUMP=Y
                                         34     *
   00003C                                35            DS    0F
   00003C 00000000                       36     PRVLEN CXD                THIS IS WHERE THE LINKAGE EDITOR STORES
   000040 00001000                       38                  =F'4096'     THE LENGTH OF THE PRV
   000044 00000001                       39                  =F'1'
   000000                                40            END   PRVINIT
```

```
(IN)     LOAD PSEUDO2
(OUT)    %  P500 PSEUDO2/000/02-86-09 LOADED
(OUT)    (MSG) %  %  E560 PSEUDO2
(IN)     AT L'32';D L'0':L'10000'
(IN)     R
(OUT)    ************** PROGRAMMENDE BEISPIEL 2 *********
(OUT)    INTERRUPTED AT 000032
(OUT)     VM-ADR 000000:010000
(OUT)     000000   05A058E0  A0E605CE  1881D223  8068A0EA  D2238044 .....W...aK.....K...
(OUT)     000014   A07FD220  8020A0A3  D21F8000  A0C40700  4510A02E ."K....tK....D......
(OUT)     000028   0000004C  00000034  0A430A5B  0A1C0700  4510A046 ...<.......$........
(OUT)     00003C   01000004  40404040  00000075  0A090000  00350000 ....    ............
(OUT)     000050   405C5C5C  5C5C5C5C  5C5C5C5C  5C5C5C40  D7D9D6C7  ************** PROG
(OUT)     000064   D9C1D4D4  C5D5C4C5  40C2C5C9  E2D7C9C5  D340F240 RAMMENDE BEISPIEL 2
(OUT)     000078   5C5C5C5C  5C5C5C5C  5CD5C9C5  40D2C1D5  D540C4C9 *********NIE KANN DI
(OUT)     00008C   C540E6C5  D3E340C9  C8D440C1  D5E3E6D6  D9E340E2 E WELT IHM ANTWORT S
(OUT)     0000A0   C1C7C5D5  4BC4D6C3  C840C7C5  D9D540C7  C9C2E340 AGEN.DOCH GERN GIBT
(OUT)     0000B4   C1E4E2D2  E4D5C6E3  40C1D3D3  C540E6C5  D3E3C1E4 AUSKUNFT ALLE WELTAU
(OUT)     0000C8   C640C6D9  C1C7C5D5  6BC4C9C5  40C5D940  D5C9C540 F FRAGEN,DIE ER NIE
(OUT)     0000DC   C7C5E2E3  C5D3D3E3  4B400000  00000110  C5C9D540 GESTELLT. ......EIN
(OUT)     0000F0   D4C5D5E2  C3C840E6  C9D9C440  D4E4C5C4  C540E2C5 MENSCH WIRD MUEDE SE
(OUT)     000104   C9D5C5D9  40C6D9C1  C7C5D57A  05B01722  5830B03A INER FRAGEN:........
(OUT)     000118   5D20B03E  5A30B042  4510B01A  01000003  04000000 )...!................
(OUT)     00012C   0A4912FF  078C0700  0A1C0700  4510B036  01010004 ...|................
(OUT)     000140   40404040  00000075  0A090000  0000008C  00001000     ................
(OUT)     000154   00000001  00000000  00000000  00000000  00000000 ....................
(OUT)    =000168   00000000  00000000  00000000  00000000  00000000 ....................
(OUT)     000FF0   00000000  00000000  00000000  00000000  C1E4C640 ................AUF
(OUT)     001004   C6D9C1C7  C5D56BC4  C9C540C5  D940D5C9  C540C7C5 FRAGEN,DIE ER NIE GE
(OUT)     001018   E2E3C5D3  D3E34B40  C4D6C3C8  40C7C5D9  D540C7C9 STELLT. DOCH GERN GI
(OUT)     00102C   C2E340C1  E4E2D2E4  D5C6E340  C1D3D3C5  40E6C5D3 BT AUSKUNFT ALLE WEL
(OUT)     001040   E3000000  D5C9C540  D2C1D5D5  40C4C9C5  40E6C5D3 T...NIE KANN DIE WEL
(OUT)     001054   E340C9C8  D440C1D5  E3E6D6D9  E340E2C1  C7C5D54B T IHM ANTWORT SAGEN.
(OUT)     001068   C5C9D540  D4C5D5E2  C3C840E6  C9D9C440  D4E4C5C4 EIN MENSCH WIRD MUED
(OUT)     00107C   C540E2C5  C9D5C5D9  40C6D9C1  C7C5D57A  00000000 E SEINER FRAGEN:....
(OUT)    =  VIRTUAL MEMORY ADDRESS 001FE0   THRU 010000   ARE NOT ALLOCATED
```

DUMMY REGISTER VECTOR

PSREG1

PSREG2

PSREG3

PSREG4

## REFERENCES


[1 ] BS2000
**DMS Tape Processing**
Reference Manual

Target group
BS2000 users, assembly language programmers
(both non-privileged).
Contents
Functions of the Data Management System in BS2000;
DMS commands and macros, service and action macros;
access methods UPAM, SAM and BTAM for tape files.
Applications
BS2000 interactive/batch mode, programming.

[2 ] BS2000
**DMS Disk Processing**
Reference Manual

Target group
BS2000 users, assembly language programmers
(both non-privileged).
Contents
Functions of the Data Management System in BS2000;
DMS commands and macros, service and action macros;
access methods UPAM, SAM, ISAM and EAM for disk files.
Applications
BS2000 interactive/batch mode, programming.

[3 ] BS2000
**LMS**
Reference Manual

Target group
BS2000 users.
Contents
Description of the statements for creating and managing program
libraries with LMS.
Applications
BS2000 interactive/batch mode.

[4 ] BS2000
**Executive Macros**
Reference Manual

Target group
BS2000 Assembler programmers (non-privileged); system
administrators.
Contents
All Executive macros in alphabetical order with detailed
explanations and examples; selected macros for DMS and TIAM;
macro overview according to application areas; comprehensive
training section dealing with eventing, serialization, inter-task
communication, contingencies.
Applications
BS2000 application programs.

[5 ] **Siemens System 7.000**
Reference Manual and Instruction List

Target group
Assembly language programmers.
Contents
Description of privileged and non-privileged instructions of the CPUs 7.500 and 7.700.
Description of the functions of these dp systems such as storage system, data formats, registers, program execution, processor states, dynamic address translation, program interrupt, input-output, error handling.

[6 ] BS2000
**Job Variables**
Reference Manual

Target group
BS2000 users.
Contents
Applications for job variables in controlling and monitoring jobs and program runs; conditional job control; all the necessary commands and macros; application examples.
Applications
BS2000 timesharing mode.

[7 ] BS2000
**System Reference Guide**
**System User**

Target group
Experienced BS2000 users.
Contents
Survey of
- commands and macros in BS2000
- instructions and assembly statements
- statements for the software products and utility routines
  - EDT, EDOR, SORT, LMS, ARCHIVE, PERCON, LEASY
  - TSOSLNK, DCAT, PASSWORD, FDEXIM, FDRIVE, DPAGE, SODUMP, TPCOMP2, PRSERVE
- essential tables and registers of BS2000
- code tables
- system standards.

Applications
BS2000 interactive/batch mode.

From

Name .....

Company .....

Street .....

City / Postal Code .....

Phone .....

Date .....

To
Siemens AG

K D ST QM 2
Manualredaktion

Otto-Hahn-Ring 6
D-8000 München 83

## Reader's Reply Form

**ASSEMBLER (BS2000)**
**Reference Manual**
**Edition May 1987**

**Order No. U62-J-Z55-6-7600**

| Page | Criticisms / Suggestions / Corrections |
|---|---|
| | |

| Page | Criticisms / Suggestions / Corrections |
| --- | --- |
| | |

From

Name .....

Company .....

Street .....

City / Postal Code .....

Phone .....

Date .....

To
Siemens AG

K D ST QM 2
Manualredaktion

Otto-Hahn-Ring 6
D-8000 München 83

## Reader's Reply Form

**ASSEMBLER (BS2000)**
**Reference Manual**
**Edition May 1987**

**Order No. U62-J-Z55-6-7600**

| Page | Criticisms / Suggestions / Corrections |
|---|---|
| | |

| Page | Criticisms / Suggestions / Corrections |
| --- | --- |
| | |

Published by Bereich Datentechnik
Postfach 830951, D-8000 München 83

Siemens Aktiengesellschaft

# BS2000

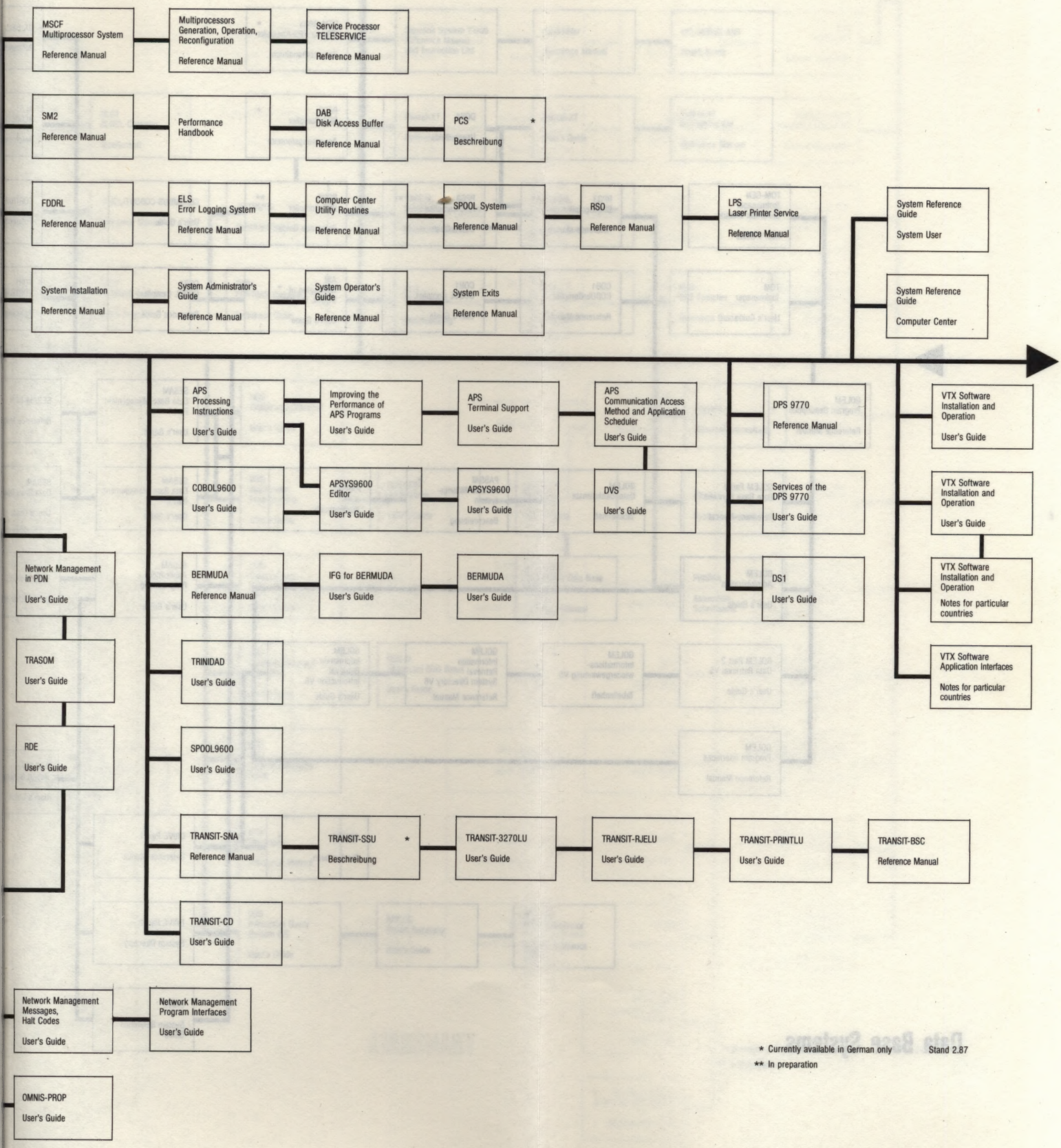
- System Messages — Reference Manual
- Systems Standards — Reference Manual
- DMS Tape Processing — Reference Manual
- DMS Disk Processing — Reference Manual
- Executive Macros — Reference Manual
- LEASY — Reference Manual
- RFA — Reference Manual
- ADAM — Reference Manual
- LMS — Reference Manual
- ARCHIVE — Reference Manual
- Utility Routines — Reference Manual
- Linkage Editor and Loaders — Reference Manual
- EDT File Editor — Reference Manual
- EDT — Statement Formats
- EDOR File Editing System — Reference Manual
- SORT Sort/Merge Routine — Reference Manual
- PERCON Conversion Program — Reference Manual
- Control System Command Language — Reference Manual
- Job Variables — Reference Manual
- AID Advanced Interactive Debugger — Reference Manual
- AID Debugging on Machine Code Level — User's Guide
- Interactive Debugging Aid (IDA) — Reference Manual
- SDF User Commands — Reference Manual
- SDF System Administrator Commands — Reference Manual
- Einführung in SDF ** für Systemverwaltung — Beschreibung
- SDF-A — Reference Manual
- SDF-A — Reference Guide
- Introductory Guide for System Users — User's Guide
- Introductory Guide for System Operators — User's Guide
- Introductory Guide to the SDF Dialog Interface — User's Guide
- Introductory Guide to Device Management — User's Guide

# TRANSDATA

- System Standards — Reference Manual
- Network Access for Terminals — User's Guide
- Datensichtstation * 9754 Schnittstelle für Programmierer — Benutzerhandbuch
- Model 9001 Printer Programmer's Interface — User's Guide
- Model 9003 Printer Programmer's Interface — User's Guide
- Model 9004 Printer Programmer's Interface — User's Guide
- Drucker 9013 * Schnittstelle für Programmierer — Benutzerhandbuch
- DCAM Program Interfaces — Reference Manual
  - DCAM COBOL Calls — User's Guide
  - DCAM Macros — User's Guide
- UTM * Planen und Entwerfen — Beschreibung
  - UTM, UTM-D Distributed Transaction Processing — Reference Manual
    - UTM Generating and Administering Applications — User's Guide
      - UTM ** Programming Applications in Pascal-XT — User's Guide
      - UTM Programming Applications in COBOL — User's Guide
      - UTM Programming Applications in PL/I — User's Guide
      - UTM-RPC1 UTM for RPG Users — User's Guide
- TIAM — User's Guide
- FHS — User's Guide
  - IFG for FHS — User's Guide
- ASMUS — User's Guide
  - ASMUS2 * — Benutzerhandbuch
- RCOM — User's Guide
  - RCOM-S — Reference Manual
    - RCOM FTP — Reference Manual
- File Transfer in Open Computer Networks — User's Guide
- Teletex-Anschluß * an BS2000 mit TTX-B — Benutzerhandbuch
- Connection to Packet Switching Networks (X.25, X.29) — Reference Manual
- Generation of a Data Communication System — User's Guide
- Network Management in BS2000 — User's Guide
- NEOSP — User's Guide
- TDA — User's Guide
- ADS — User's Guide
- RBAS — User's Guide
- OMNIS — User's Guide

# Publications Plan for

- BS2000
- TRANSDATA
- Programming Systems
- Data Bases

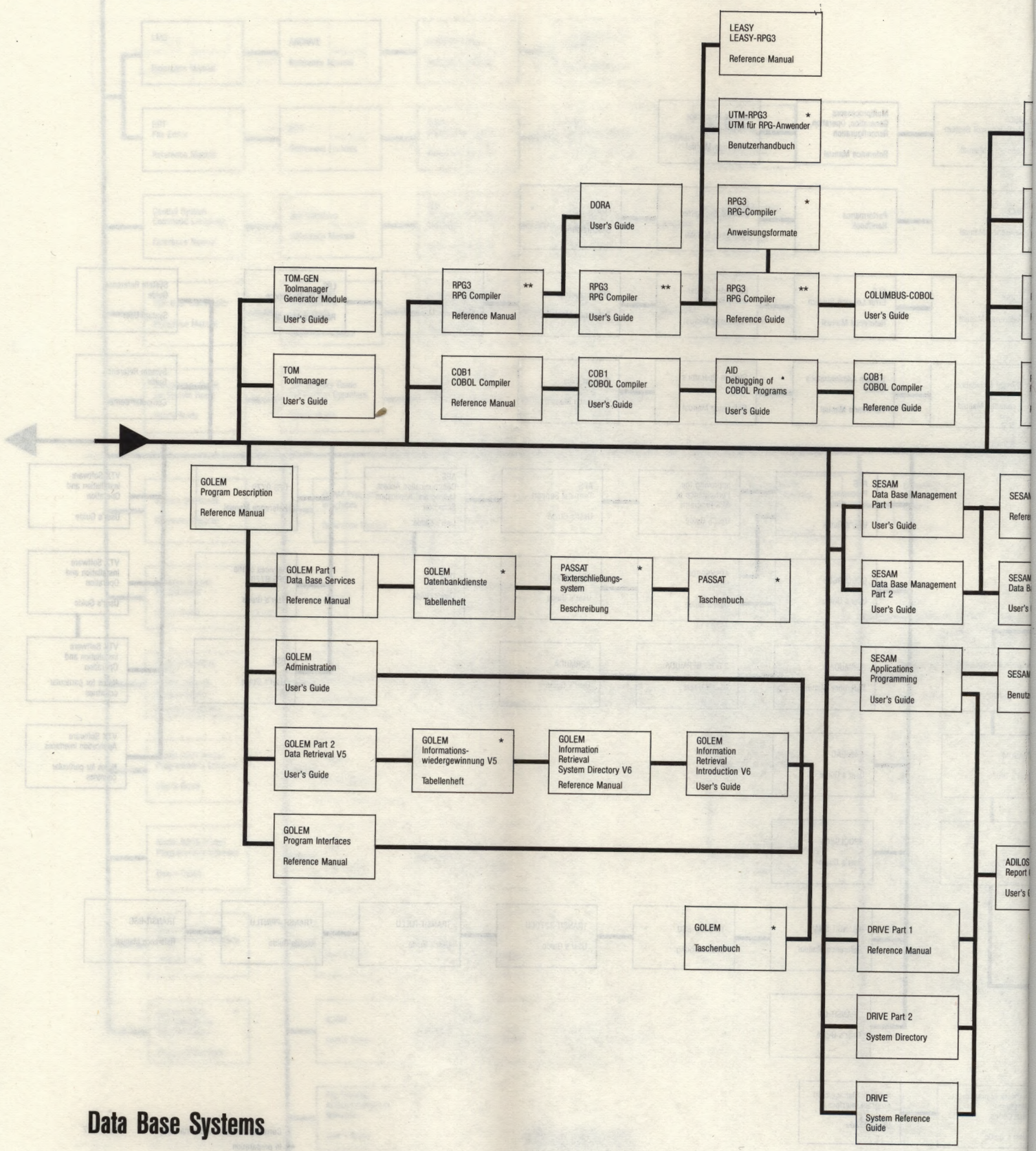



* Currently available in German only

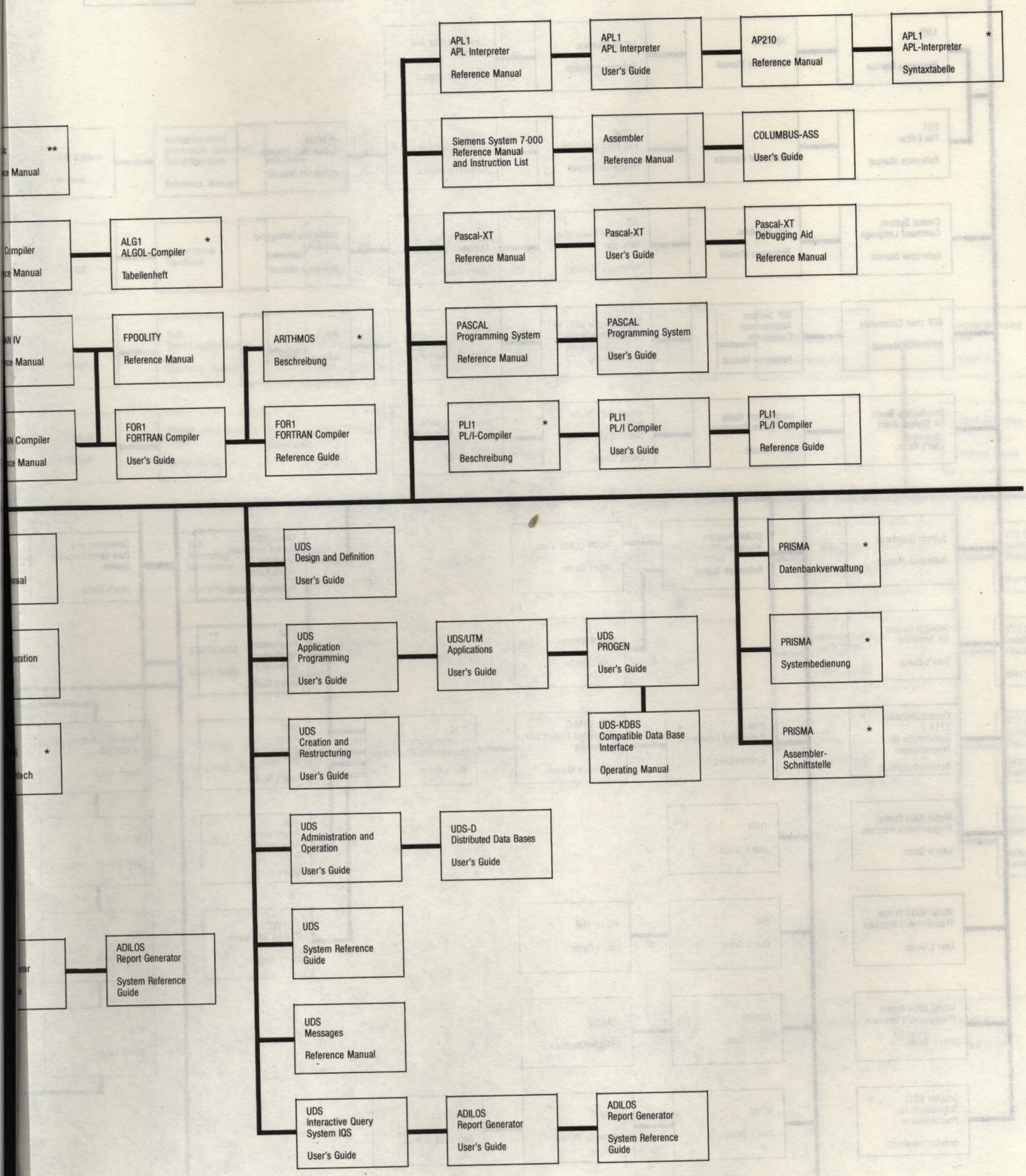
** In preparation

Stand 2.87

# Programming Systems




# Data Base Systems




* Currently available in German only

** In preparation

Stand 2.87